uc3m | Universidad Carlos III de Madrid

Grado Ingeniería Electrónica Industrial y Automática
Curso académico 2018-2019

Trabajo Fin de Grado

"Implementación de red neuronal en FPGA"

Javier Moreno Segurado

Tutor

Honorio Martín González

Leganés, 4 de Julio de 2.019.

Agradecimientos

Me gustaría dar las gracias en primer lugar, a mi tutor Honorio Martín, por su gran ayuda y consejos en la realización de este trabajo. También, agradecer a toda mi familia y amigos, en especial a mis padres por todo el apoyo incondicional que me han dado.

Sin todos ellos, este proyecto no hubiese sido posible.

Resumen

En la actualidad las redes neuronales están cobrando cada vez más importancia, siendo más frecuente su uso en diversos campos como: la robótica, finanzas, medicina o marketing. Las redes neuronales son un conjunto de algoritmos desarrollados normalmente en lenguaje software. La idea surge en un intento de imitar el funcionamiento del cerebro, creando un sistema capaz de aprender por sí mismo.

Gracias a este aprendizaje automático, las redes neuronales se han convertido en un campo muy importante de la inteligencia artificial, destacando en tareas relacionadas con la visión artificial.

El aumento en el uso de este tipo de sistemas deriva en la búsqueda de nuevos dispositivos y técnicas, donde poder ser implementados como la implementación hardware. En este contexto, se empieza a investigar en la implementación de redes neuronales en una FPGAs. Una FPGA, es un circuito electrónico de lógica programable. Este tipo de dispositivos, también están adquiriendo cada día más importancia en diversos sectores, siendo común su uso en campos como: la automoción, defensa y el sector aeroespacial.

Este trabajo surge con la idea de combinar ambos conceptos, realizando la implementación de una red neuronal en una FPGA. La red diseñada será capaz de identificar el digito representado en una imagen, mediante el reconocimiento de patrones. Para ello, en primer lugar, la red será diseñada y entrenada en lenguaje software. Posteriormente con los datos obtenidos durante el entrenamiento se realizará la implementación hardware. Analizando los resultados obtenidos, con la finalidad de conocer si es posible y tiene sentido implementar una red neuronal en este tipo de dispositivos.

Abstract

Nowadays, neural networks are becoming more important, being used in different areas such as: robotics, economics, medicine or marketing. Neural networks are a combination of algorithms usually developed in software language. The idea comes up, during an attempt of replicating the brain function, creating a system capable of learning by itself.

Thanks to this machine learning, neural networks turned into in a very important field of artificial intelligence, standing out in tasks related with computer vision. The rising use of these kind of systems, lead to the research of new devices and implementation techniques, such as hardware implementation. In this context, the study of neural network implementation on an FPGA began. A FPGA is a logic programable electronic circuit. This kind of devices are becoming increasingly important in areas like automation, defense and space.

This work comes up with the idea of performing a neural network implementation on a FPGA. The designed network will be able to identify the handwritten digit represented on an image, by means of pattern recognition. For doing that, the first thing would be to create and train a neural network in software language. Afterwards, with the data obtained from the training process, the hardware implementation would be performed. By analyzing the obtained results, we would be able to know if it is possible to implement a neuronal network in this kind of devices and if doing that makes sense.

Índice

Índice de figuras

Índice de tablas

Listado de abreviaturas y acrónimos

FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
ANN	Artificial Neuronal Network
VHDL	VHSIC Hardware Description Language
ReLU	Rectifier Linear Unit
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
MNIST	Modified National Institute of Standards and Technology
IP	Intellectual Property
IEEE	Institute of Electrical and Electronics Engineers
RAM	Random Access Memory
UART	Universal Asynchronous Receiver – Transmitter
DSP	Digital Signal Processor
XDC	Xilinx Design Constrains
LUT	Look Up Table
MIF	Memory Initialization File

1. Introducción

1.1. Importancia de las redes neuronales

La evolución de la tecnología, principalmente la creación y el desarrollo de los ordenadores y microprocesadores, nos han llevado a una era en la que se busca crear sistemas inteligentes, cada vez más independientes, capaces de aprender por sí mismos, tomando como referencia a los seres humanos.

En este contexto, se desarrollan las redes neuronales, las cuales, están adquiriendo más importancia día a día, llegando a ser encontradas en diversos ámbitos con la capacidad de influir en la vida diaria de las personas. Una red neuronal artificial, también conocida, bajo las siglas ANN (Artificial Neural Network), es un conjunto de algoritmos cuya finalidad es resolver un problema específico, siendo capaces de extraer las características más importantes de los distintos datos de entrada y aprender mediante ejemplos. Surgen como una inspiración biológica, en un intento de modelar el funcionamiento del sistema nervioso [1].

Su uso puede estar destinado a infinidad de campos, desde la robótica, la economía o las finanzas. Las redes neuronales destacan a la hora de resolver problemas asociados con el reconocimiento de patrones, siendo capaces también de ofrecer una solución para problemas relacionados con pronósticos y predicciones, así como problemas de control y optimización, entre otros, llegando a convertirse en un campo muy importante de la inteligencia artificial. Gracias a la cantidad de soluciones que son capaces de aportar, su uso se extiende a diversas áreas, ya que sirven de ayuda en campos como la medicina permitiendo desarrollar aplicaciones las cuales permiten mejorar o salvar la vida de diversas personas. A su vez, permiten desarrollar otras aplicaciones en áreas como en la informática clasificando y analizando bases de datos, en marketing recomendando anuncios basados en intereses de los usuarios, en distintas páginas webs o incluso las finanzas. La característica principal es que estos sistemas tienen la capacidad de aprender por sí mismos y reaccionar correctamente a las distintas situaciones que se puedan encontrar.

Su área por excelencia es el reconocimiento de imágenes, sirviendo de gran ayuda en tareas relacionadas con la visión artificial. Lo que les permite reconocer y clasificar los distintos patrones que se encuentran en una imagen. Llegando a desarrollar aplicaciones como los coches autónomos, los cuales pueden usar esta tecnología para la detección y reconocimiento de señales, paneles indicadores u obstáculos. El reconocimiento de imágenes también tiene una gran utilidad en áreas como el reconocimiento facial, o reconocimiento de firmas con el fin de evitar posibles estafas.

1.2. ¿Dónde se puede implementar una red neuronal?

Como se ha mencionado anteriormente, estos dispositivos aparecen en nuestra vida cotidiana con una frecuencia cada vez mayor, lo que nos lleva a preguntarnos ¿Qué dispositivos son capaces de ejecutar una red neuronal?

Al ser un modelo informático, la mayoría de las redes desarrolladas actualmente están programadas en lenguaje software, lo que les permite ser implementadas comúnmente en microprocesadores, procesadores o unidades de procesamiento gráfico, también conocidas bajo las siglas GPU (Graphics Proccesing Unit). Todos estos dispositivos, son unidades de procesamiento en las cuales se ejecutan aplicaciones software. La diferencia más notable es que la GPU, está compuesta por miles de núcleos capaces de ejecutar tareas en paralelo. Sin embargo, un procesador está compuesto por uno o varios núcleos que funcionan de forma secuencial.

Actualmente, también podemos encontrar programas o aplicaciones diseñados principalmente para ordenadores, que permiten crear y hacer uso de una red neuronal, sin la necesidad de que el usuario tenga que programar y elaborar el código. Estos programas incluyen la posibilidad de crear varios tipos de redes, dependiendo de su fin, permitiendo al usuario la posibilidad de entrenar la red con los datos que considere oportunos. Un ejemplo de este tipo de software es el programa Matlab, el cual incluye una herramienta para elaborar redes neuronales.

Alternativamente, también se está investigando en el desarrollo de redes neuronales implementadas en lenguaje hardware, lo que nos permite la posibilidad de implementar la red en una FPGA (Field Programable Gate Array). Una FPGA es un dispositivo de lógica programable, es decir, un circuito electrónico completamente configurable capaz de realizar una tarea, para la que ha sido programado. Estos dispositivos están siendo cada vez más usados, llegando a encontrar aplicaciones específicas para las cuales es necesario hacer uso de una FPGA.

La posibilidad de implementar una red neuronal en una FPGA supone un gran avance, tanto como para el desarrollo de las FPGA como en el ámbito de la inteligencia artificial, debido a que da lugar a nuevos dispositivos, con distintas características y cualidades diferentes a los dispositivos, donde eran implementadas estas redes habitualmente. El conjunto de motivos, mencionados anteriormente, da lugar a la motivación de realizar este trabajo e intentar realizar una implementación hardware de una red neuronal.

A continuación, se procederá a explicar las ventajas y desventajas que nos podemos encontrar a la hora implementar una red neuronal en los distintos dispositivos mencionados anteriormente.

1.2.1. Ventajas del uso de FPGA

Las ventajas del uso de una FPGA, frente a los dispositivos software son la siguientes:

La implementación software, suele estar limitada por el tiempo que tardan en ejecutarse los distintos algoritmos. No obstante, este tiempo se consigue reducir en los dispositivos GPU, debido a su paralelismo, lo que les permite la posibilidad de ejecutar un programa desarrollado en software más rápido, que un procesador o microprocesador.

A su vez, las GPU presentan un consumo de potencia mucho mayor si se compara con una FGPA o un microprocesador. Este elevado consumo de potencia ocasiona, que las GPU tengan altas temperaturas de funcionamiento y es necesario que incorpore un sistema de refrigeración, lo que encarece el precio del dispositivo y su flexibilidad a la hora de ser adaptadas en un sistema.

Sin embargo, una FPGA es un dispositivo reconfigurable creado para realizar una tarea, generalmente de manera muy eficiente, la implementación hardware es más rápida que la implementación software, principalmente debido al paralelismo que presentan este tipo de dispositivos, al ser un circuito.

Por otro lado, si miramos el precio de adquisición de una unidad de procesamiento gráfico, es mucho mayor que cualquier otro dispositivo, el precio de una GPU de gama media, capaz de ejecutar una red neuronal puede estar comprendido entre los 400€ y 600€, con un consumo aproximado de 300W [2].

El precio de adquisición de una FPGA capaz de ejecutar una red neuronal, puede estar comprendido entre 100€ y 200€ con un consumo aproximado entre 2.5W y 9W. Datos obtenidos de la comparativa realizada por la empresa Berten, dedicada al procesamiento de señales digitales [2].

1.2.2. Desventajas del uso de FPGA

Debido a que los dispositivos GPU son los más potentes, se ha decidido centrarse en las desventajas frente a estos dispositivos.

La principal desventaja a la hora de implementar una red neuronal en una FPGA es la capacidad procesamiento, la cual es mucho menor que una GPU, principalmente a la hora de operar con datos en coma flotante, también conocidos por su acrónimo en inglés como floating point. A su vez tienen una capacidad de procesamiento inferior que los procesadores para realizar ciertas tareas.

Por otro lado, los dispositivos GPU cuentan con acceso a memoria directo, DMA (Direct Memory Access) lo que significa que no es necesario un ciclo de reloj cada vez que se

quiere acceder a la memoria. Una FPGA normalmente necesita un ciclo de reloj para acceder y escribir los distintos datos en una memoria.

En cuanto al precio de adquisición, los dispositivos software suelen ser más caros que una FPGA. Salvo en el caso de los microprocesadores, el precio de compra microprocesador es mucho menor que el de una FPGA. Podemos llegar a encontrar un microprocesador por un precio de un de 20€.

1.2.3. Resumen de la comparativa

Como conclusión de esta comparación, podemos obtener que la capacidad de procesamiento de una GPU es mayor comparado con una FPGA; sin embargo, el precio de adquisición de una FPGA es mucho menor y a su vez ofrece un menor consumo energético. Por estas razones las FPGA pueden ser una alternativa eficiente a la hora de realizar determinadas tareas, en este caso para la ejecución de redes neuronales que no requieran una gran capacidad de procesamiento, el uso de una FPGA puede ser una solución muy factible.

1.3. Objetivo

El objetivo principal de este trabajo es realizar una implementación en lenguaje hardware de una red neuronal.

La red neuronal a implementar será una réplica de una red neuronal desarrollada y entrenada en software. Más concretamente se ha optado por replicar una red neuronal creada utilizando el programa Matlab y su herramienta para la creación de redes neuronales.

Para probar la funcionalidad de dicha red neuronal, se ha elegido una red neuronal orientada al reconocimiento de patrones. Específicamente, la finalidad de la red creada es el reconocimiento de dígitos del 0 al 9 en una imagen de 28x28 píxeles.

La red deberá ser capaz de reconocer que número aparece representado en la imagen, la dificultad consiste en que los números representados en las imágenes corresponden a fotografías de números escritos manualmente, por distintas personas con caligrafías diferentes.

Para llevar a cabo dicha tarea se utilizará una placa de desarrollo de la marca Zybo, modelo Zynq -7000, la programación de dicha placa de hará mediante lenguaje VHDL (VHSIC Hardware Description Language).

Teniendo en cuenta este objetivo general, se han establecido los siguientes objetivos más específicos:

- **Diseño, entrenamiento y test de la red neuronal en Matlab** para el reconocimiento de patrones. Esto consiste en ser capaz de diseñar y entrenar en Matlab una red que cumpla los objetivos establecidos, documentándose e investigando todo lo posible sobre el diseño de redes neuronales.

- **Implementación funcional en FPGA**, el cual es objetivo principal de proyecto. Con los datos obtenidos en Matlab al hacer el entrenamiento se debe crear una red neuronal en hardware.

- **Mejorar la eficiencia de la red**, reduciendo todo lo posible el número de ciclos de reloj. Una forma de llevar a cabo esta solución es realizar de manera más efectiva los distintos procesos, aprovechando el paralelismo ofrecido por estos dispositivos realizando varias tareas al mismo tiempo.

- **Cálculo de las funciones mediante tablas de aproximación**, esto consiste en crear otro proyecto aparte, el cual tendrá la misma finalidad, pero a diferencia de la red creada anteriormente, el cálculo de las funciones de activación se realizará mediante tablas de aproximación, también conocidas por su nombre en inglés como look up tables. Esta mejora nos permitirá ahorrar el cálculo de la función de activación, creando una red más eficaz, debido a que se reducirá el tiempo de cálculo y los recursos utilizados.

Una vez se haya alcanzado todos estos objetivos, se recopilarán y analizarán todos los resultados.

1.4. Estructura del documento

A continuación, en este apartado se explicarán y describirán cada uno de los capítulos que componen este trabajo:

- **Capítulo 1, Introducción:** En este capítulo se explicará la definición de una red neuronal, describiendo los principales dispositivos usados para la implementación de redes neuronales, así como los inconvenientes de cada dispositivo a la hora de realizar dicha implementación. Por último, se describirán los objetivos que se pretenden cumplir en este trabajo.

- **Capítulo 2, Estado del Arte:** Se hará una mención a la historia de las redes neuronales, explicando cómo surgieron. Seguidamente, se explicarán los conceptos básicos de una red neuronal, conceptos como su funcionamiento, composición, los principales tipos y aplicaciones de redes que podemos encontrar en la actualidad. Por último, se detallarán las distintas características a tener en cuenta para realizar una implementación en hardware. Así como las implementaciones que se han podido llevar a cabo en estos dispositivos y la normativa utilizada en este proyecto,

- **Capítulo 3, Solución Propuesta:** Este capítulo está reservado, a explicar los pasos a seguir para poder lograr el objetivo propuesto anteriormente. En primer lugar, se detallará el caso de estudio y después se procederá a explicar paso a paso la solución escogida para realizar la implementación y las respectivas mejoras, y el presupuesto necesario para la realización de este proyecto. En último lugar, se mostrarán los resultados obtenidos y el análisis de dichos resultados.

- **Capítulo 4, Conclusión:** Se analizará el trabajo realizado, se expondrán las distintas conclusiones obtenidas del análisis de los resultados y las futuras ampliaciones que se podrían llevar a cabo.

2. Estado del Arte

El estado del arte trata de explicar el desarrollo de las redes neuronales, su funcionamiento o los distintos tipos que se pueden encontrar, así como sus diversas aplicaciones en la actualidad.

2.1. Historia

El origen de las redes neuronales se remonta a 1.943, donde el médico Warren McCullock y el matemático Walter Pitts, con la intención de desarrollar inteligencia artificial, tomando como referencia el funcionamiento del cerebro, presentan por primera vez el primer modelo matemático de una red neuronal. Este modelo sirvió para sentar las bases de las redes neuronales, pero no fue hasta 1.958 cuando el psicólogo Frank Rosenblatt creó el perceptrón, lo que se conoce como la unidad fundamental de las redes neuronales que podemos encontrar en la actualidad [3] .

Como se ha mencionado anteriormente, el desarrollo de las redes neuronales surge gracias a la idea de imitar el funcionamiento del sistema nervioso humano, en concreto el celebro. Para poder entender esta inspiración biológica, deberemos explicar brevemente el funcionamiento del sistema nervioso. El cerebro está formado por pequeñas células llamadas neuronas, las cuales son el componente elemental del sistema nervioso. Cada neurona, es capaz de recibir y transmitir información a las neuronas contiguas, mediante impulsos eléctricos.

Las neuronas están compuestas principalmente por tres partes bien diferenciadas el núcleo, las dendritas y los axones. El núcleo es el encargado de procesar la información, del cual surgen prolongaciones ramificadas llamadas dendritas y axones. Mediante las dendritas las neuronas son capaces de recibir mensajes de otras neuronas, mientras que la transmisión de información tiene lugar en los axones. La transmisión de información entre las distintas neuronas recibe el nombre de sinapsis, a la hora de producirse esta sinapsis, no se tiene en cuenta únicamente la información transmitida por una neurona aisladamente. Si no, que cada neurona recibe la información procedente del resto de neuronas y realiza una suma algebraica de toda la información obtenida [4].

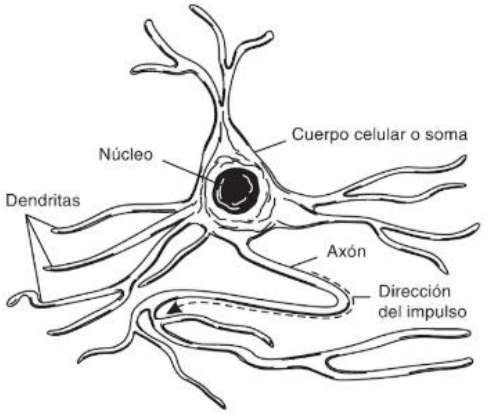


Figura 1: Composición de una neurona [3].

Una vez explicado el funcionamiento del sistema nervioso, podemos ver como sirve de inspiración e influencia en la creación de las redes neuronales artificiales. Donde el perceptrón corresponde a lo que sería la neurona, estos perceptrones se agrupan entre sí de la misma forma que ocurre en el cerebro.

En el cerebro podemos observar una asociación de un gran número de neuronas, las cuales se agrupan y comunican entre sí, llegando a formar un conjunto capaz de procesar y reaccionar ante esa información. La importancia reside en el valor de ese conjunto entero, una sola unidad carece de significado, la unión de todas las unidades trabajando en conjunto permiten analizar y dar sentido a la información recibida [4].

Para poder desarrollar una respuesta consistente, es necesario realizar una suma aritmética, teniendo en cuenta el peso o valor de cada información recibida. Esto mismo podemos encontrarlo en las redes neuronales artificiales, cada perceptrón almacena la información que le llega de los perceptrones contiguos, asignando a cada información un peso o ponderación, definiendo el valor de esa información recibida, tal y como veremos en el siguiente capítulo con más detenimiento.

2.2. Fundamentos básicos

En este apartado, se van a explicar los fundamentos básicos de las redes neuronales, para poder entender su correcto funcionamiento y comprender el trabajo realizado. Como ya hemos explicado anteriormente, una red neuronal es un modelo matemático capaz de resolver un problema mediante aprendizaje automático, también conocido como aprendizaje de las máquinas o machine learning, en inglés. Las redes neuronales, se basan en una pequeña unidad llamada perceptrón, la idea reside en que está pequeña unidad sea capaz de almacenar información y transmitir esa información a los demás perceptrones, llegando a formar una densa red la cual recibe el nombre de red neuronal.

Este perceptrón, es la unidad fundamental de la red, podría decirse que es el equivalente a una neurona en el caso del cerebro, y por ese motivo el perceptrón también es conocido como neurona artificial. Matemáticamente un perceptrón es un elemento, cuya finalidad es almacenar información, normalmente suele contener un número definido entre 0 y 1. Los perceptrones, de una red se encuentran conectados entre sí, de modo que, cada perceptrón puede estar formado por un número indefinido de entradas y salidas.

Los perceptrones o neuronas se agrupan formando capas, la mayoría de las redes neuronales que podemos encontrar son multicapas, lo que significa que están formadas por más de una capa. A la hora de examinar una red neuronal, la primera capa que nos podemos encontrar suele obtener el nombre de capa de entrada, en inglés se le conoce como Input Layer y, en esta capa podemos encontrar las neuronas de entrada. Estas neuronas son las encargadas, de recibir los datos de partida para resolver el problema. Por ejemplo, en el caso de una red neuronal encargada de procesar los datos que se encuentran en una imagen, las neuronas de entrada recibirán el valor de los píxeles que componen dicha imagen. La capa de entrada posee un número indeterminado de neuronas, normalmente dependiendo del número de entradas que sean necesarias para resolver el problema.

Cada una de estas neuronas de entrada, establece una conexión con cada neurona de la siguiente capa, la cual recibe el nombre de capa oculta, también conocida como hidden layer por su traducción en inglés. La capa oculta, puede ser simplemente una capa o puede estar formada por un conjunto indefinido de capas, dependiendo de las necesidades del problema a resolver.

Por último, cada una de las neuronas de la última capa oculta, se comunica con las neuronas la siguiente capa, siendo la última capa de la red llamada capa de salida u output layer. En esta capa se muestra el resultado de la red, dependiendo del problema a resolver cada una de estas neuronas corresponde con cada una de las salidas del problema.

Como ya se ha mencionado anteriormente, estos perceptrones o neuronas almacenan un número comprendido normalmente entre 0 y 1. En las neuronas de salida, ese número nos indicará las probabilidades de que esté activa la salida que representa dicha neurona. En el caso particular de nuestra red, si la neurona de salida, encargada de representar el número 3 tiene un valor de 0.9, significa que, según la red, la imagen introducida tiene un 90% de posibilidades de ser un 3.

En la figura 2, podemos encontrar una ilustración que refleja la arquitectura de una red neuronal multicapa, indicando cada una de las capas que componen una red neuronal. En la imagen podemos observar que cada neurona perteneciente a una capa está conectada con todas las neuronas de la capa contigua.

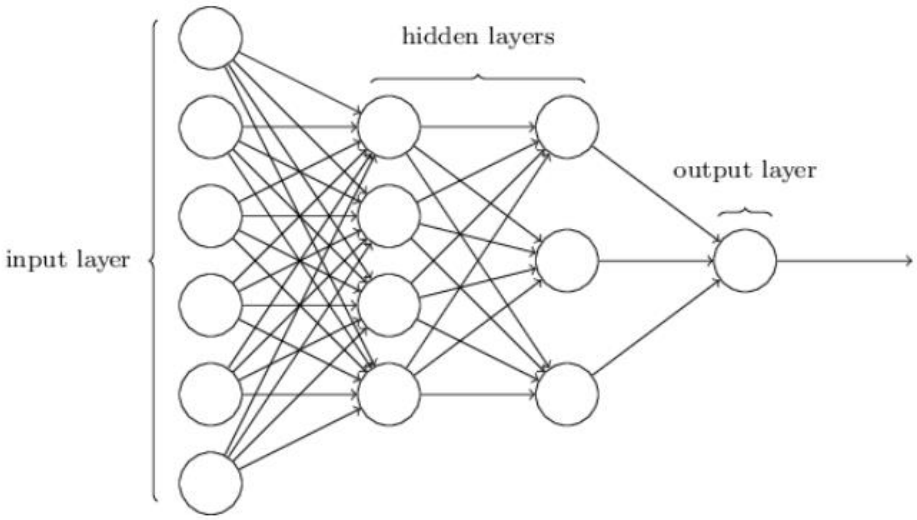


Figura 2: Esquema de la distribución de capas de una red neuronal [5].

Las únicas neuronas que tienen un valor inicial son las neuronas de entrada, el valor del resto de las neuronas se irá calculando a medida que se ejecute la red. Para hallar la solución de la red no se tiene en cuenta, únicamente la información proporcionada por una neurona, sino que es una contribución de todas ellas. El cálculo de las neuronas de las capas ocultas y la capa de salida se obtiene mediante una ecuación matemática, representada en la ecuación 1.

Para calcular el valor de la primera neurona de la capa oculta, por ejemplo, el procedimiento a seguir por la red será el siguiente:

El valor de la primera neurona de entrada es multiplicado por un peso, cada conexión realizada entre dos neuronas recibe un peso sináptico o weight, con la finalidad de ponderar la información. Al resultado de esta multiplicación se suma el valor guardado en la neurona que estamos tratando de calcular, en este caso, el valor de la primera neurona de la capa oculta. Debido a que es la primera operación realizada el valor guardado sería 0, una vez terminada esta operación se guardaría el valor obtenido en la primera neurona de la capa oculta.

Una vez haya sido guardado el valor, se pasa a la segunda neurona de la capa de entrada, realizando el mismo proceso, de modo que se realiza un sumatorio de todas las neuronas de entrada multiplicadas por su peso correspondiente. Una vez haya tenido lugar la suma de todas las neuronas multiplicadas por los pesos, al resultado se le sumará un bias. El bias, es un número que definirá cuanto le costará a la neurona obtener un valor de 1, si a una neurona se le suma un bias muy grande, esa neurona tendrá más facilidades para tener un valor de 1 o cercano a 1. Sin embargo, si a la neurona se le suma un bias negativo será más difícil para esa neurona obtener un valor de 1.

Por último, a la neurona se le aplica una función de activación, la finalidad de esta función de activación es acotar los valores de la neurona entre 0 y 1. Toda esta explicación se

puede resumir mediante la ecuación matemática que podemos encontrar en la ecuación 1.

$$N = \sigma\left(\sum_{i=1}^{n} x_i \cdot \omega_{ji} + b\right)$$

Ecuación 1: Ecuación para el cálculo de una neurona.

Donde N es la neurona que queremos calcular, x representa el valor de las neuronas ubicadas en la capa anterior y la letra ω representa los distintos pesos. El subíndice i indica el número de neuronas que se encuentran en la capa anterior y el subíndice j corresponde al número de la posición en la que se encuentra la neurona, que está siendo calculada en su capa correspondiente. Por último, b representa al bias y σ a la función de activación correspondiente.

Una vez realizados todos estos cálculos podemos obtener el valor final de una neurona, en el caso de ejemplo, el valor de la primera capa oculta, el cálculo del resto de neuronas de esa capa oculta se realiza de la misma forma. Para rellenar las capas sucesivas tanto capas ocultas como la capa de salida, el procedimiento a seguir será el mismo únicamente en vez de multiplicar las neuronas de entrada por un peso, se multiplicará las neuronas de la última capa calculada. Como vemos, esto forma una especie de cadena donde el resultado de la última capa nos ayuda a calcular el resultado de la capa donde nos encontramos, esto es conocido como feedfoward.

En cuanto a la función de activación, el principal objetivo es que el valor final de la neurona sea normalmente, un resultado comprendido entre 0 y 1. Podemos encontrar varios tipos de funciones de activación, dependiendo de la función elegida el valor de neurona convergerá hacia un valor igual a 1, o igual a 0 de una forma más rápida o más lenta. La función de activación más conocida es la sigmoide, también conocida como la función logística, pero podemos encontrar muchas otras como la ReLU, la cual está presente en un gran número de redes neuronales.

En la tabla 1, podemos encontrar los tipos más comunes de funciones de activación con sus correspondientes derivadas. Como podemos observar, no todas las funciones tienen como resultado valores comprendidos entre 0 y 1, podemos encontrar algunas particularidades como la función tangente hiperbólica. Las funciones que más rápido convergen, suelen ser las funciones lineares como la función identidad(identity), o funciones similares como la ReLu o PReLU, las cuales podemos encontrar líneas rectas con diferente pendiente dependiendo si x es mayor o menos que 0. La función Binary Step, en español conocido como función escalón también converge muy rápido debido a cuando x es igual a 0 el valor de la función pasa a ser directamente a 1.

Por otro lado, podemos encontrar funciones que convergen más lento como la sigmoide, en la tabla 1 la podemos encontrar como función logística, en inglés logistic function, o la tangente hiperbólica, representada en la tabla bajo su abreviatura Tanh.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Tabla 1: Funciones de activación más comunes [6].

La razón por la que las redes neuronales son consideradas una rama de la inteligencia artificial, es debido a que poseen la capacidad de “aprender”. Por ello para terminar está sección de fundamentos básicos se procederá a explicar brevemente en que consiste el entrenamiento de una red neuronal.

El aprendizaje es la manera en la que se halla el valor de los pesos y bias utilizados en la ejecución de la red. Existen diferentes formas de realizar el entrenamiento de una red neuronal, generalmente el entramiento está formado por una serie de algoritmos que permiten cambiar los pesos y bias de la red, de forma automática. Cuando se realiza el entrenamiento de una red, se probará la red con distintos datos de entrada, mediante una serie de algoritmos la red adaptará los pesos y bias para que el resultado de la red concuerde con el introducido.

Cuantas más veces se realice el entrenamiento de la red y cuanto más sean distintos datos de entrada, más robusta será la red a la hora de resolver el problema y tendrá mayores probabilidades de existo. Existen distintos métodos de entrenamiento, dependiendo si el

aprendizaje es supervisado o no, según el método elegido la red variará sus pesos y bias de una forma u otra, el más común es el método de retro propagación, conocido como backpropagation.

2.3. Tipos de redes neuronales

A la hora de clasificar las redes neuronales, podemos hacerlo principalmente mediante dos formas, mediante su **topología** o según la forma en la que realizan **el aprendizaje**.

2.3.1. Clasificación según su topología

Trata de clasificar las redes dependiendo de su arquitectura, según la distribución de las capas y teniendo en cuenta la forma en la que se distribuyen las conexiones entre las neuronas de distintas capas que forman la red, podemos encontrar varios tipos:

- **Redes monocapa**: Esta categoría está formada por redes las cuales están compuestas de una única capa, en esa única capa podemos encontrar un número indefinido de neuronas. Perteneciente a esta categoría podemos encontrar redes como la red de Hopfield.

Red de Hopfield: Como podemos observar en la figura 3, es una red compuesta por una única capa donde todas las neuronas están conectadas entre sí. Al ser una única capa las neuronas de entradas están directamente conectadas con las neuronas de salida. Normalmente las neuronas de esta red toman valores comprendidos entre -1 y 1, a la hora de ejecutar la red, el valor de las neuronas debe actualizarse repetidas veces hasta que converge en un valor adecuado.

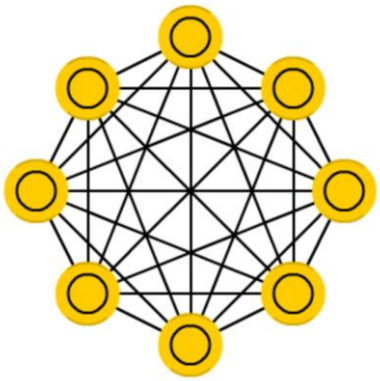


Figura 3: Red de Hopfield [7].

- **Redes multicapa:** Comprende a las redes neuronales, cuya arquitectura esté formada por más de una capa, donde todas las neuronas de cada capa estén conectadas a todas las neuronas de la capa contigua. Es la categoría más común, a la cual pertenecen la mayoría de las redes con las que nos podemos encontrar.
  La arquitectura y funcionamiento de este tipo de redes está descrita en el apartado anterior de fundamentos básicos. En la figura 2, podemos encontrar un esquema de la distribución de este tipo de redes.

- **Red neuronal convolucional**: Este tipo de redes se pueden asemejar a las redes multicapa con la diferencia de que no todas las neuronas están conectadas entre sí. La arquitectura es la siguiente:

  Al inicio de la red podemos encontrar capas convolucionales y capas de reducción las cuales se van alternando; finalmente, podemos encontrar capas que presentan una conexión total entre sus neuronas como en una red multicapas. Las capas convolucionales, se encargan de filtrar los datos de entrada, multiplicando la capa de entrada por un filtro de kernel. Este filtro, es una pequeña matriz la cual se irá multiplicando por pequeños grupos de neuronas de entrada mediante un producto escalar. Las neuronas de entrada se dividen en matrices de las mismas dimensiones que el filtro de kernel, las cuales serán multiplicadas por este filtro, con la finalidad de obtener determinadas características de la capa de entrada. Normalmente, después de cada convolución se aplica la función de activación ReLU.

  A continuación, encontramos la capa de reducción que se encarga de disminuir la dimensión de la red, conservando la información más relevante, esto es realizado de forma estadística con ayuda de ecuaciones como el promedio o el valor máximo. Por último, encontramos la capa de clasificación, la cual tiene la misma topología y funciona de la misma manera que una red multicapa. En la figura 4, podemos encontrar un esquema de la distribución de capas y la arquitectura que presentan este tipo de redes.

  El objetivo de este tipo de redes es reducir el número neuronas necesarias, centrándose en las características más relevantes de la capa de entrada. De modo que la capa de clasificación tendrá un número de neuronas menor gracias a este filtrado y reducción, por lo que será necesario un menor número de operaciones y datos a manejar; lo que implica, una menor capacidad de procesamiento a la hora de realizar los cálculos. Presentan gran utilidad en redes con un gran de número de neuronas en la capa de entrada, como es el caso de las imágenes, la cuales suelen tener una neurona por píxel, por lo que su uso es muy frecuente a la hora de analizar imágenes o tareas relacionadas con la visión artificial [8].

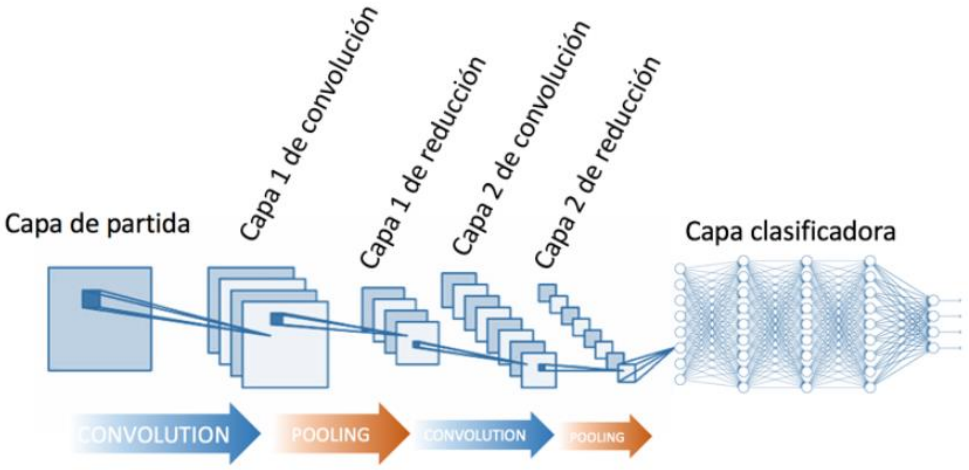


Figura 4: Red neuronal convolucional [8].

- **Red neuronal recurrente**: Este tipo de redes no se agrupan en capas ordenadas, sino que cada neurona presenta conexiones aleatorias con el resto de las neuronas. Las redes recurrentes destacan por tener una retroalimentación, es decir, en la entrada de la neurona se realiza una suma de la información obtenida en ese instante más el resultado que contiene neurona. Por lo tanto, se tiene en cuenta la información proporcionada a la entrada y la información obtenida a la salida de esa misma neurona la cual había sido calculada anteriormente.

  Esto nos permite decir, que, en cierto modo, estas redes tienen memoria, debido a que tienen en cuenta el resultado anterior. La figura 5, representa una neurona recurrente simple, donde podemos observar la retroalimentación mencionada anteriormente y la forma en la que la neurona va guardando la información con el paso del tiempo. Donde S representa la neurona en el estado actual, S_{t-1} el instante anterior y S_{t+1} el próximo.

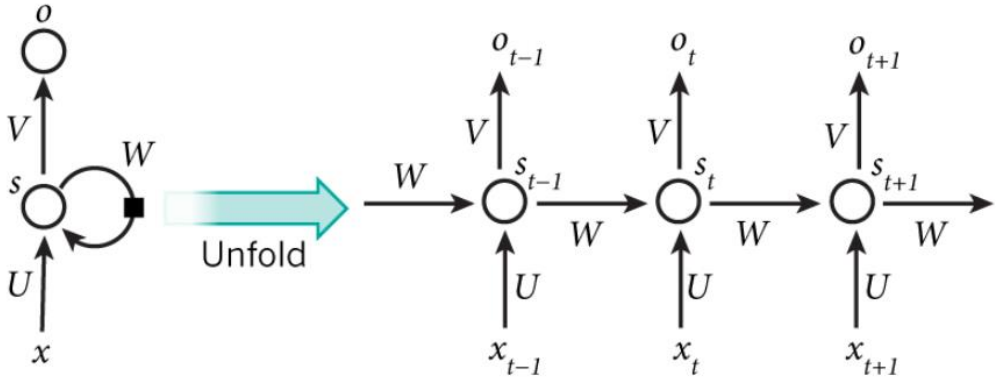


Figura 5: Red neuronal recurrente simple [9].

Podemos encontrar varios tipos de redes recurrentes, dependiendo de la arquitectura formada los más comunes son:

- Red LSTM (Long short-term memory): Cuya traducción al español sería memoria a corto plazo. En esta red podemos encontrar en la capa intermedia lo que se conoce como bloques de memoria. Estos bloques son neuronas las cuales se encuentran retroalimentadas, siendo capaces de almacenar la información durante un periodo de tiempo.

  Los bloques de memoria presentan tres puertas, encargadas de controlar el proceso, estas puertas son: Puerta de entrada, controla la información que entra en la memoria. Puerta de salida, se encarga de regular la salida de información cuando sea necesario su uso. Puerta de olvido, permite borrar parte del contenido de la memoria, dejando espacio libre para información relevante. Son usadas para tareas como reconocimiento vocal, reconocimiento de gestos o tareas donde se maneje un conjunto amplio de datos.

  En la figura 6, podemos encontrar un esquema de una red LSTM donde input y output, corresponde a las capas de entrada y salida respectivamente. La puerta de entrada se encuentra representada bajo la letra x_t, la puerta de olvido como f_t y la puerta de salida como o_t [10].

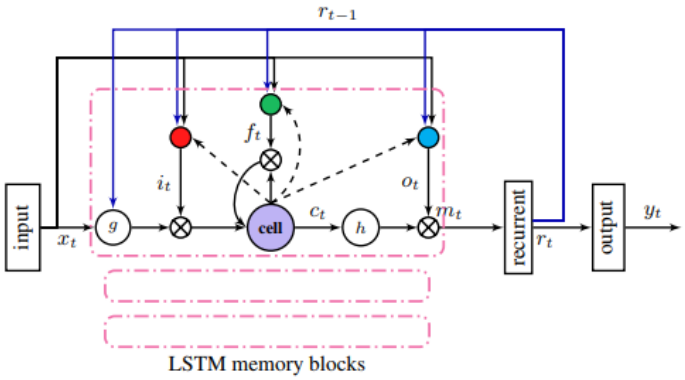


Figura 6: Red Long short-term memory (LSTM) [10] .

- Red GRU (Gated recurrent unit): Este tipo de red es similar a las redes LSTM, con la diferencia que el flujo de información dentro de la unidad es controlado mediante dos puertas, que son: Puerta de reajuste, permite controlar los datos de entrada que se van a leer, dando la posibilidad de elegir la cantidad de datos provenientes de la salida que se van a conservar y olvidar. Puerta de actualización: controla cual va a ser el contenido que se va a mantener en la celda.

En la figura 7, podemos observar un esquema de una Gated recurrent unit, donde r representa la puerta de reajuste, z la puerta de activación, h representa la información contenida en el bloque y $\tilde{h}$ representa la información nueva [11]. Pueden ser de gran utilidad en campos similares en los que podríamos encontrar redes LSTM, con la diferencia que las redes GRU son más efectivas cuando se maneja un conjunto pequeño de datos.

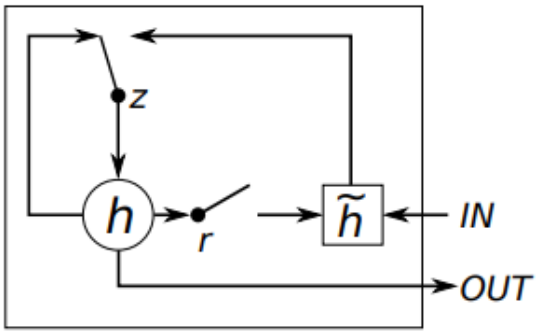


Figura 7: Gated Recurrent Unit (GRU) [11].

2.3.2 Clasificación según el aprendizaje

Esta clasificación pretende diferenciar los distintos tipos de redes según la forma en la que realicen el entrenamiento. Podemos encontrar dos tipos aprendizaje supervisado y aprendizaje no supervisado.

- **Aprendizaje supervisado:** Cuando se realiza el entrenamiento de este tipo de redes se introducen los distintos datos de entrada y a su vez se introducen en la red los resultados deseados. Dicho de otra forma, a la red se le comunica según el dato de entrada introducido cual es la salida que debe mostrar. El tipo de aprendizaje supervisado más común es el algoritmo de retropropagación o backpropagation.

Backpropagation: Recibe este nombre, debido a que, a partir de los resultados obtenidos en el entrenamiento, modifica los pesos y bias empezando por la última capa oculta hasta llegar a la capa de entrada.

Al ser un método de aprendizaje supervisado se conoce el resultado de la entrada con la que se está entrenando, por lo que este método se basa en calcular el error obtenido y ajustar los distintos pesos y bias en función de este error. Para ello, lo primero que se debe hacer es calcular la función de coste o error cuadrático medio. En la ecuación 2, podemos ver representada esta función de coste. Donde y representa el valor obtenido a la salida de la neurona y x es el valor que deseamos alcanzar, 1 si la neurona debe estar activada 0 si se da el caso contrario.

$$E = \frac{1}{2} \cdot \sum_{i=1}^{n} (x - y)^2$$

Ecuación 2: Función de coste.

Una vez obtenida la función de coste, se calcula el gradiente negativo de esta función de coste. Esto nos dará lo que se conoce como gradiente de descenso, el gradiente muestra el camino más rápido según el cual, el error sería el máximo, al ser negativo muestra el camino más rápido para obtener el mínimo error. Con ayuda de este gradiente y una constate α, la cual recibe el nombre de learning rate, podemos ajustar los pesos y bias para minimizar el error, lo que nos permite mejorar el resultado obtenido por la red [5].

- **Aprendizaje no supervisado:** Al realizar el entrenamiento simplemente se introducen en la red los distintos tipos de datos de entrada, sin ninguna información sobre el resultado deseado. Los datos de entrada introducidos a la hora de hacer el entrenamiento, no tienen asociados su respectiva salida.

  Podemos hacer una subclasificación dependiendo del modo en el que se realiza este aprendizaje no supervisado.

  - Análisis de los componentes principales: Este tipo de aprendizaje se basa en realizar un análisis de las propiedades encontradas en los datos de entrada, mediante una serie de algoritmos. Los cuales tratan de identificar como esas propiedades identificadas se relacionan entre sí y las influencia que ejercen unas sobre las otras.

  - Aprendizaje competitivo: En las redes competitivas, las distintas neuronas aprenden a analizar conjuntos de patrones, de modo que compiten entre sí activándose la neurona cuyos pesos tengan un mayor parecido con el patrón introducido. La neurona ganadora, refuerza sus conexiones, de modo que los pesos de la unidad ganadora tengan una mayor similitud con el patrón de entrada [12].

  - Aprendizaje por refuerzo: Por último, tenemos el aprendizaje por refuerzo, el cual en algunas ocasiones está considerado dentro del grupo de aprendizaje no supervisado, pero en otras ocasiones se puede encontrar como una clasificación a parte. Corresponde a un tipo de red que comparte características con ambos tipos de aprendizajes, supervisado y no supervisado. No llega a formar parte del aprendizaje supervisado, debido a que no se dispone de la salida exacta que debe aparecer en la red, pero si se comprueba si el patrón de salida es el adecuado. Este último tipo de aprendizaje está basado en premiar las conductas adecuadas de la red,

cuando se obtiene una salida similar a la que queremos y reforzar negativamente cuando no ocurre esto. Este refuerzo se realiza variando los pesos sinápticos de la red.

2.4. Aplicaciones de las redes neuronales

Las redes neuronales es un área, el cual está en expansión, día a día van surgiendo más utilidades y aplicaciones para este tipo de inteligencia artificial, llegando a estar más presentes en nuestra vida diaria. Principalmente debido a su capacidad de aprendizaje y su capacidad para procesar la información recibida, siendo capaces de seleccionar determinadas características o patrones. Lo que hace que cada vez sea más común el uso de este tipo de inteligencia artificial en un sinfín de tareas y diferentes campos, no solo la robótica o la automática. A continuación, se explicarán algunas áreas en las cuales se hace uso de redes neuronales.

2.4.1. Procesamiento de imágenes

Una de las aplicaciones más importantes de las redes neuronales es el procesamiento de imágenes. Las redes neuronales nos permiten analizar y clasificar las distintas imágenes, siendo capaces de extraer la información que se encuentra en ellas. Lo cual, nos permite distinguir los distintos números representados en una imagen o distinguir el objeto que representado. Por ejemplo, una red neuronal sería capaz de distinguir si en la imagen introducida aparece un perro o una persona, o un gato, siendo posible clasificar las distintas imágenes en grupos.

A su vez, destacan por su habilidad para identificar y clasificar los distintos patrones presentes en diferentes imágenes, lo cual tiene una gran utilidad como reconocimiento facial, reconocimiento de huellas dactilares o el reconocimiento de firmas.

Este campo tiene muchas aplicaciones muy importantes como el desarrollo de visión artificial pudiendo ser utilizada en campos como robótica como la medicina. En los últimos años, las redes neuronales están adquiriendo cada vez más importancia en este campo desempeñando tareas como el diagnóstico de enfermedades, gracias a la capacidad de las redes neuronales para el análisis de imágenes, pueden ser utilizadas para analizar radiografías, ecografías o resonancias magnéticas, con el fin de detectar ciertas enfermedades como, por ejemplo, infartos o ciertos tumores.

También, cabe destacar, que presentan la capacidad de realizar diagnósticos a partir de los resultados de un análisis de sangre, pudiendo detectar enfermedades como el Alzheimer [13].

2.4.2. Predicciones

Las redes neuronales presentan una gran capacidad a la hora de hacer predicciones, la mayoría de estas predicciones son elaboradas con fines financieros, y son realizadas mediante el algoritmo de backpropagation.

Su uso en este campo se debe a que las redes neuronales son capaces de predecir series temporales caóticas, lo cual tiene aplicaciones financieras muy importantes, como en el mercado de valores para predecir el precio de las acciones.

También, presenta una utilidad muy importante para los bancos a la hora de aprobar créditos, predecir futuros cambios en las distintas divisas, o analizar transacciones con el fin de evitar fraudes.

2.4.3. Control y fusión sensorial

Es muy común encontrar redes diseñadas para tareas relacionas con la Ingeniería de Control, aprendiendo mediante métodos de aprendizaje supervisado, siendo capaces de a reaccionar a las distintas entradas y salidas, lo que nos permite crear controladores como PID.

Hay que destacar, que este tipo de redes puede ser usadas para tareas como la fusión sensorial. Frecuentemente en campos como la robótica, podemos encontrar multitud de sensores, aportando cada uno distinta información, siendo la fusión sensorial la encargada de procesar este conjunto de datos. El uso de redes neuronales nos permite procesar la información de cada sensor, obteniendo las características más relevantes.

2.4.4. Otras aplicaciones

También podemos encontrar diversas aplicaciones para las redes neuronales, las cuales podrían ser:

- **Conversión de texto a voz**: Es posible convertir el texto escrito en fonemas y con ayuda de un sintetizador generar la voz. La ventaja de usar redes neuronales para este proceso reside en la capacidad de aprendizaje de esto tipo de redes, gracias a esto no sería necesario programar ningún tipo de reglas como se realiza habitualmente. A su vez, las redes neuronales son capaces de llevar a cabo el proceso contrario, realizando un reconocimiento automático de voz posteriormente transformar esos fonemas en texto.

- **Procesamiento de señales:** Las redes neuronales también destacan a la hora de realizar un procesamiento de señales. Dentro de este campo podemos encontrar diversas tareas donde las redes neuronales pueden ser de gran utilidad como el modelado de sistemas, las redes neuronales son capaces de aprender la función de transferencia necesaria mediante la cual el sistema se comporte de forma lineal como el sistema que está modelando.

  También pueden ser capaces de servir como un filtro de ruido, eliminando el ruido que podemos encontrarnos en una señal [14].

Como podemos observar podemos encontrar las redes neuronales en diversos campos, siendo capaces de desempeñar tareas que incluso pueden salvar o mejorar la vida de las personas.

Gracias a la combinación de estos diversos campos se pueden desarrollar aplicaciones como el coche autónomo, el cual podría hacer uso de redes neuronales para el reconocimiento de imágenes, detectar patrones como las líneas de la carretera, analizar y distinguir entre los distintos tipos de señales de tráfico o hacer uso de la fusión sensorial para combinar toda la información ofrecida por los distintos sensores que se encuentran equipados en el vehículo.

De la misma forma, se dan usos muy similares de las redes neuronales en robótica, por ejemplo, en robot humanoides encargados de interactuar con humanos, podrían ser de utilidad en tareas como: el reconocimiento de imágenes, de patrones, así como la conversión de texto a voz o la fusión sensorial.

2.5. Implementaciones en una FPGA

Como se ha mencionado anteriormente, las redes neuronales son algoritmos que podemos encontrar en diferentes dispositivos, debido a que en este trabajo se centra en la implementación de una red neuronal en una FPGA. A continuación, se procederá a comentar las implementaciones que se han logrado hacer en este tipo de dispositivos.

La implementación de las redes neuronales en FPGAs se remonta a finales de la década de los 1.980 y principios de 1.990, siendo hoy en día un campo todavía en desarrollo.

La principal característica por la que se empiezan a introducir redes neuronales en estos dispositivos es el paralelismo. Esta característica nos permite que se estén realizando varias a tareas al mismo tiempo, lo que es de gran ayuda al implementar una red neuronal, permitiendo que la red pueda ser ejecutada en un tiempo menor.

Cuando se implementa una red neuronal en una FPGA, hay que tener en cuenta que tiene ciertas limitaciones. Principalmente, a la hora de realizar operaciones como la multiplicación de los pesos o al implementar ciertas funciones de activación. A su vez,

presentan limitaciones para guardar todos los pesos si la red tiene grandes dimensiones; también, depende de la forma en la que se represente el tipo de datos coma fija o coma flotante, también conocido como fixed point o floating point. Representar datos en coma fija, requiere menos recursos, pero se obtiene una menor precisión. A su vez el espacio de almacenamiento ocupado depende del tipo de datos escogidos para realizar dicha red 32 bit o 64 bit.

Lo más común a la hora de realizar este tipo de implementaciones es realizar una red multicapa, siendo entrenada mediante el algoritmo de backpropagation. En cuanto a las funciones de activación, pueden plantear ciertos problemas a la hora de realizar una implementación en hardware, principalmente aquellas que no son lineares, teniendo que realizar operación por operación, una de las funciones más común es la sigmoide. Mediante está configuración, ha sido posible crear redes encargadas del reconocimiento de voz, reconocimiento de caracteres en imágenes o patrones como huellas dactilares.

En la actualidad, es posible implementar prácticamente casi cualquier red neuronal, es una FPGA. Siendo la tarea más importante buscar la FPGA más adecuada, que aporte la eficiencia deseada o un precio razonable [15].

El fabricante de FPGA más conocido es Xilinx, el cual tiene en venta una gran variedad de productos distribuidos en varias ramas. Actualmente, Xilinx tiene en venta la serie 7 de sus productos, los cuales según sus prestaciones y precio se dividen en varias ramas o familias. Por un lado, tenemos la gama Artix, las FPGAs pertenecientes a esta familia se caracterizan por tener un precio muy económico con unas bajas prestaciones comparado con las demás familias. Por otro lado, tenemos la gama intermedia, la cual obtiene el nombre de Kintex, esta gama se caracteriza por tener un precio mayor que la gama Artix, debido a que podemos encontrar un mayor número de celdas, así como de DSP (Digital Signal Processor), lo cual implica un mayor número de módulos aritméticos y bloques de memoria RAM, que reciben el nombre de BRAM (Block Random Access Memory). Por último, podemos encontrar la gama Virtex la cual tiene un alto precio, debido a que cuenta con las más altas prestaciones.

2.6. Normativa y marco regulador

Al ser un campo aún en desarrollo, hoy en día no hay una normativa específica sobre el uso de redes neuronales. Actualmente, se están desarrollando distintos estándares por el Comité Internacional de Estandarización, también conocido como ISO (International Organization of Standarization). Dentro de esta organización existe un comité, que recibe el nombre de ISO/IEC JTC 1/SC 42, encargado de redactar distintos estándares en el campo de la inteligencia artificial. Dentro de estos estándares podemos encontrar la norma **ISO/IEC AWI 23053,** aún en desarrollo que pretenderá ser el marco regulador para aquellos sistemas basados en machine learning como es el caso de las redes neuronales.

Aunque no haya una normativa específica para el uso de redes neuronales, se debe tener en cuenta, la normativa y estándares aplicados en las distintas áreas donde nos podemos encontrar una red neuronal. Un ejemplo, podría ser el caso de los coches autónomos o la visión artificial.

La visión artificial o visión por computador se encuentra regulada por el comité ISO/IEC JTC 1/SC 24. El estándar **ISO 7942-1:1994** es el encargado de normalizar el almacenamiento, procesamiento y la modificación dinámica de imágenes en 2 dimensiones. Existen también normas específicas que hay que cumplir si en el procesamiento de imágenes está involucrado la vida de una persona. Por ejemplo, el estándar **ISO 20380:2017** regula aquellos sistemas de visión artificial encargados de detectar el ahogamiento de personas en piscinas, estableciendo las pruebas y requisitos que deben cumplir para garantizar la seguridad.

Los coches autónomos hacen uso de las redes neuronales, principalmente para el reconocimiento de señales de tráfico u obstáculos. En los vehículos autónomos, los detectores de señales de tráfico (como podría ser una red neuronal), se encuentran regulados bajo la norma **ISO 10711:2012**, cuya principal función es estandarizar los mensajes que envían los detectores al controlador.

3. Solución propuesta

A continuación, se detallarán los pasos seguidos, explicando cada una de las decisiones tomadas para llegar a alcanzar la resolución de este trabajo.

3.1. Caso de estudio

Tal y como se ha mencionado antes, el objetivo de este proyecto es implementar una red neuronal en una FPGA, para realizar el entrenamiento de la red se hará uso del programa Matlab. Por ello, lo primero que debemos hacer es crear la red capaz de resolver el problema propuesto en Matlab y posteriormente entrenarla, nos servirá como caso de estudio para poder realizar la implementación en hardware.

La red deberá ser capaz de reconocer el dígito mostrado en una imagen. Los dígitos serán números del 0 al 9, escritos en puño y letra por diferentes personas. Las imágenes introducidas en la red tendrán unas dimensiones de 28 x 28 píxeles. Esto es debido, a que para el entrenamiento de la red se ha usado la base de datos MNIST, que recibe estas siglas debido a su nombre en inglés Modified National Instute of Standards and Technology.

La base elaborada por el Instituto Nacional de Estándares y Tecnología americano es ampliamente conocida en el campo de la inteligencia artificial y machine learning. Esta base de datos contiene más de 60.000 imágenes de dígitos escritos a mano, cada uno por diferentes personas, desde niños hasta personas adultas. Debido a esta diversidad entre las distintos participantes esta base de datos es una excelente fuente para realizar el entrenamiento de una red neuronal. En la base de datos podemos encontrar el valor de cada píxel que forma la imagen, dando lugar a un total 784 píxeles, debido a que la imagen es un cuadrado de 28 píxeles de lado. El valor de estos píxeles se encuentra en escala de grises, representada mediante un número del 0 a 255, donde 255 sería que el píxel tiene un color negro y 0 cuando el color que podemos encontrar en el píxel es blanco.

Matlab dispone de una aplicación para crear redes neuronales, está aplicación nos da la opción de crear distintos tipos de redes dependiendo de la tarea que debe desempeñar la red, siendo posible encontrar redes recurrentes, multicapa o convolucionales.

En este caso, se ha escogido crear una red neuronal diseñada para el reconocimiento de patrones, debido a que la red debe ser capaz de reconocer el número escrito. Una forma de llevar a cabo esta tarea puede ser identificando los patrones más característicos que forman cada número siendo posible distinguir entre los distintos dígitos. Por ejemplo, sabemos que el número 1 suele estar formado por una línea vertical, mientras que el 2 presenta un semicírculo en la parte superior, y una línea horizontal en la parte inferior.

Distinguir los distintos patrones presentados en la caligrafía de los números, nos permite poder clasificarlos correctamente.

Una vez seleccionado el tipo de red a utilizar, Matlab nos da la posibilidad de escoger la cantidad de capas ocultas, la cantidad de neuronas en cada capa y la forma en la que se va a realizar el entrenamiento, lo que nos permite personalizar la red al gusto del usuario. Cuando se haya terminado de diseñarla red, el último paso sería introducir los datos con los que se desea entrenarla en este caso la base de datos MNIST. Debido a que se va a realizar un aprendizaje supervisado, es necesario introducir los datos de entrada, el valor de los píxeles que forman la imagen, así como indiciar el número representado en cada imagen el cual deberá ser el resultado mostrado por la red. Completado estos pasos se realizaría el entrenamiento y la res estaría lista para ser ejecutada, las características, así como el diseño escogido serán explicadas en el siguiente apartado.

3.2. Red neuronal implementada software

Para el reconocimiento de imágenes lo más común es usar una red multicapa o una red convolucional, en el caso de que la imagen posea gran resolución. En este caso, la red creada será una red multicapa, la cual contará con una sola capa oculta. Principalmente se ha escogido la red multicapa debido a que es capaz de desempeñar perfectamente la tarea y por su simplicidad, comparado con otro tipo de redes como una red convolucional.

El número de neuronas en la capa de entrada de la red viene dado por el número de entradas que tenga el problema a resolver. En nuestro caso, debido a que la entrada es una imagen de 28 x 28 píxeles, son necesarias 784 neuronas en la capa de entrada, cada una contendrá el valor de cada píxel que compone la imagen.

Debido a que en la base de datos MNIST el valor de píxel está reflejado en escala de grises, los números presentes en cada neurona de entrada tienen un valor comprendido entre 0 y 255, por lo que es necesario realizar un tratamiento previo de esos datos. Esto consiste en multiplicar el valor en la escala de grises por una ganancia, también conocida como gain en inglés y posteriormente restarle uno, la ganancia tiene un valor comprendido entre 0 y 1. Siendo el valor de cada píxel multiplicado por un gain distinto, este preprocesado de los datos es realizado automáticamente por Matlab. En la ecuación 3, aparece representada la ecuación, donde x es el valor del píxel en escala de grises y G el gain proporcionado por Matlab.

$$y = G \cdot x - 1$$

Ecuación 3: Ecuación para el preprocesado de datos.

En cuanto a las capas ocultas, el número de capas ocultas dependerá de si los datos son linealmente separables. En el caso de que lo sean, sólo será preciso una capa oculta. En el caso de que los datos no sean linealmente separables, se deberá hacer uso de un mínimo de 2 o 3 capas, siendo importante elegir el número de capas exactas. Debido a que, si se introducen muchas capas ocultas, se necesitaría una mayor capacidad de procesamiento y un mayor tiempo para ejecutar la red. Si uno no está seguro de la relación que guardan sus datos de entrada, lo mejor es empezar con el mínimo posible e ir añadiendo capas si el resultado de la red no es el correcto.

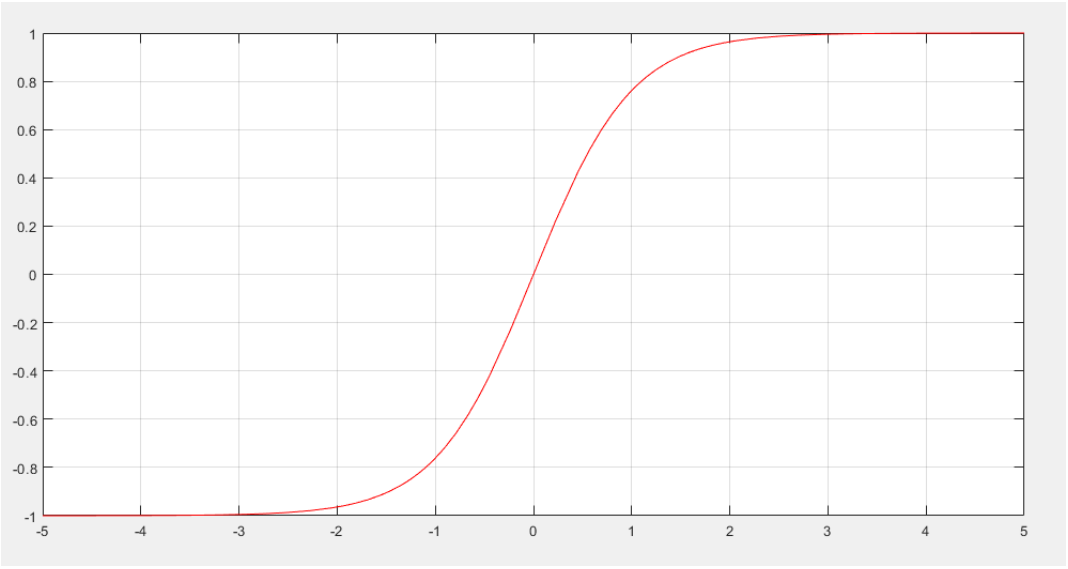
En relación con el número de neuronas, tampoco hay ninguna regla exacta que nos permita calcular el número de neuronas necesarias. Si el número de neuronas es menor que el imprescindible, la red no podrá obtener una solución correcta al problema; independientemente de cuantas veces se realice el entrenamiento. Esto se conoce como underfitting. También, puede suceder el caso opuesto llamado overfitting, teniendo lugar cuando el número de neuronas presentes en la capa es mayor que el necesario. Lo que produce una especialización de las neuronas mayor de la requerida; es decir, aprenden demasiado sobre un determinado tipo de patrón, siendo muy difícil obtener una solución correcta, en el caso de que los datos difieran de este patrón aprendido.

La mejor solución para hallar el número necesario de neuronas en una capa es mediante prueba y error, empezar por el menor número de neuronas posibles y comprobar los resultados obtenidos tras el entrenamiento; si el resultado obtenido no es el adecuado, se deberá aumentar el número de neuronas. Podemos encontrar diferentes fórmulas, que nos permiten calcular el número de neuronas por el que comenzar a realizar las pruebas, pero no existe ninguna garantía de que sea el método adecuado. Se recomienda, que, en ningún caso, el número de neuronas de una capa oculta sea mayor que la mitad del número de neuronas presentes en la capa de entrada [16].

Debido a que la finalidad de este trabajo es implementar la red en lenguaje hardware, se ha optado por reducir todo lo posible, tanto el número capas ocultas, como el número de neuronas en dichas capas. Con la intención de reducir lo máximo posible la capacidad de procesamiento necesaria, así como los recursos utilizados. Debido a que como se ha mencionado anteriormente, es una de las limitaciones con las que nos podemos encontrar a la hora de utilizar en una FPGA. Se ha optado por empezar con una sola capa oculta y comprobar si los resultados obtenidos eran los correctos. Para hallar el número de neuronas de esa capa, se han realizado diversas pruebas, llegando a la conclusión de usar 10 neuronas en esa capa oculta, dado que es el menor número de neuronas con el que se ha logrado obtener unos resultados satisfactorios.

El número de neuronas en la capa de salida depende de las posibles soluciones que pueda tener el problema a resolver. El objetivo de este proyecto es reconocer los distintos dígitos del 0 al 9, representados en imágenes. Por ello serán necesarias 10 neuronas en la capa de salida, y cada neurona corresponderá a un número del 0 al 9.

En relación con las funciones de activación, en un principio la intención era usar la función sigmoide, debido a que no converge excesivamente rápido, lo cual implicar una mayor precisión, debido a que tenemos una mayor resolución a la hora de acotar el resultado entre 0 y 1. A su vez, es una de las más usadas en las redes multicapa. Finalmente, dado a que el entrenamiento de la red ha sido realizado en Matlab, no ha sido posible escoger el tipo de función de activación, porque Matlab tiene fijada una función específica para el reconocimiento de patrones, la tangente hiperbólica. En la figura 8, podemos encontrar una imagen de la tangente hiperbólica y su ecuación correspondiente.



$$\tanh = \frac{2}{1 + e^{-2 \cdot x}} - 1$$

Figura 8: Función tangente hiperbólica

Como podemos observar, la función tangente hiperbólica es un caso particular, debido a que abarca valores comprendidos entre -1 y 1. Esto no presenta ningún problema, a no ser que está función se presente en la capa de salida. Si se hace uso de esta función en la capa de salida, se podrían tener dificultades para obtener la solución al problema, dado que los valores obtenidos a la salida no serán acotados entre 0 y 1. Lo que implica una mayor dificultad a la hora de conocer que neurona se encuentra activada en la capa se salida. Como se ha mencionado anteriormente, no es obligatorio que las neuronas tengan un valor comprendido entre 0 y 1, aunque es lo más común. En la figura 8, se puede apreciar que la función tangente hiperbólica converge más rápido que la función sigmoide y más lento que funciones como la ReLU, por lo que podría considerarse como un término intermedio.

La función de activación mencionada únicamente será utilizada en la capa oculta, en la capa de salida se hará uso de otra función de activación conocida como softmax. La

función softmax, presenta ciertas particularidades comparada con otras funciones de activación, pero su uso es muy útil, sobre todo en la capa de salida. Debido a que tiene en cuenta los resultados de todas las neuronas de la capa de salida, siendo posible calcular las posibilidades que tiene cada neurona de ser el resultado del problema, en forma de porcentaje.

Para poder entender correctamente la función softmax, la podemos descomponer en distintas partes la primera parte, consiste en realizar una función exponencial de cada neurona de salida. Una vez que se haya realizado esta función exponencial en todas las neuronas de salida, se llevará a cabo un sumatorio de los resultados. Este sumatorio estará compuesto por la suma de todas las exponenciales de salida, por lo que se puede nombrar como suma total. El último paso que realizar, será dividir la exponencial de cada neurona entre esta suma total, lo que nos permite conocer la importancia de cada neurona sobre esa suma total en forma de porcentaje. En la ecuación 4, aparece representada la ecuación de la función softmax, donde x el resultado de la neurona antes de ser aplicada la función, j es el número de neuronas en la capa de salida, e i indica la posición de la neurona se salida que se va a calcular.

$$Softmax = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Ecuación 4: Función softmax.

A pesar de estar basada en una función exponencial, la cual no nos permite acotar los valores entre 0 y 1. Los valores finales si quedan acotados debido a que se normaliza el resultado de cada exponencial.

La figura 9, representa la estructura de la red creada en esta implementación software. Donde aparecen el número de neuronas en cada capa, así como las distintas funciones de activación en el caso de la capa oculta la función tangente hiperbólica y en el caso de la capa de salida, la función softmax.

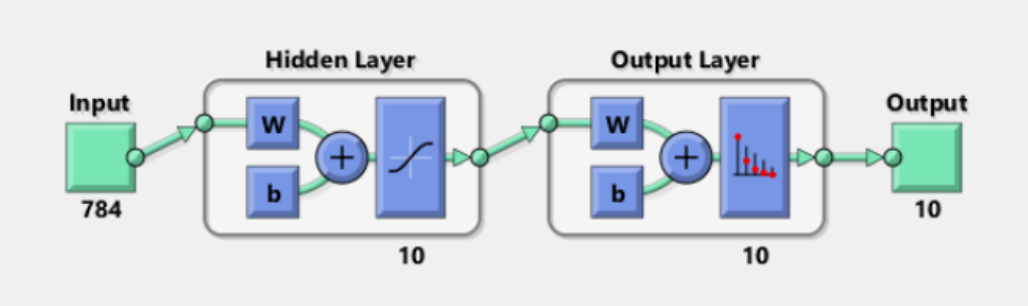


Figura 9: Estructura de la red.

Por último, en relación con el entrenamiento de la red, se ha realizado un entrenamiento mediante el algoritmo de backpropagation. Podemos encontrarlo brevemente explicado en el capítulo sobre estado del arte, como ejemplo de aprendizaje supervisado. Matlab se encargará de realizar el entrenamiento con los datos introducidos, la base de datos MNIST, y una vez haya sido finalizado el entrenamiento, obtendremos cada uno de los pesos para las distintas conexiones que tiene la red, así como los bias.

Cada neurona de entrada se conecta con todas las neuronas de la capa oculta, estableciendo un total de 10 conexiones por cada neurona de entrada. Dado que tenemos 784 neuronas de entrada y 10 en la capa oculta, obteniendo un total de 7840 conexiones entre la capa de entrada y la capa oculta. Las neuronas de la capa oculta se conectarán de la misma forma con las neuronas de la capa de salida debido a que tenemos 10 neuronas en la capa de salida las conexiones presentes serán igual a $10 \cdot 10 = 100$ conexiones entre la capa oculta y la capa de salida. Formando un total de 7940 en toda la red, lo que implica que podemos encontrar un total de 7940 pesos. En relación con el bias, encontramos un bias distinto cada vez que se calcula el resultado final de una neurona, debido a que tenemos 10 neuronas en la capa oculta y 10 en la capa de salida, obtenemos un total de 20 bias distintos.

3.3. Implementación en FPGA

En esta solución propuesta, el programa diseñado será capaz de reconocer el dígito que se encuentra representado en una imagen. Mostrando el número reconocido mediante 4 luces leds, ubicadas en la FPGA, estos leds compondrán la representación binaria del número reconocido.

Para desarrollar la implementación hardware, ha sido utilizado el programa Vivado HLx versión 2016.4., el cual es un software creado por Xilinx para el desarrollo hardware en lenguaje VHDL o verilog, para su posterior implementación en una FPGA de la misma marca. En este trabajo, se ha hecho uso del lenguaje VHDL para la programación de la FPGA.

Las entradas utilizadas de la FPGA serán las siguientes: por un lado, se hará uso del reloj ubicado en FPGA, el cual funcionará a una frecuencia de 100MHz. A su vez será necesario un interruptor el cual será usado para resetear el programa y un botón para iniciar la ejecución de la red. Las salidas utilizadas serán las 4 luces Leds encargadas de representar el dígito reconocido.

El diseño de la red se realizará, mediante un diagrama de bloques, antes de comenzar la explicación a la solución propuesta, es necesario tener en cuenta algunas características, como la representación y tamaño de los datos a manejar.

Los datos manejados por la FPGA serán representados en punto flotante o floating point con un tamaño de 32 bits. Debido a que la hora de realizar los distintos cálculos se obtiene una mayor precisión en coma flotante, respecto a los datos representados en coma fija. A

su vez, para realizar los diversos cálculos se ha utilizado una IP ya creada, la cual trabaja en floating point. Una IP, es un módulo, creado con anterioridad por otra persona, e integrado en el programa Vivado, el cual nos permite realizar ciertas tareas, sin la necesidad de desarrollar el código y a su vez incluye la posibilidad de ser configurado por el usuario. Recibe este nombre, debido a que son las sigas de Intelectual Property, en español propiedad intelectual. Este bloque presenta gran utilidad a la hora de realizar operaciones como: multiplicaciones, divisiones o funciones exponenciales, pero es necesario que los datos estén representados en coma flotante para poder realizar las operaciones. Todas las distintas entidades o módulos utilizados en este programa han sido diseñados en lenguaje VHDL, excepto las IP, que ya están integradas en el programa.

En cuanto al tamaño de los datos hay dos posibilidades: 32 o 64 bits. Como se ha mencionado anteriormente, una de las limitaciones de las FPGAs a la hora de implementar una red neuronal, es la memoria utilizada debido a la cantidad de datos que hay que almacenar. Por este motivo, se ha escogido un tamaño de 32 bits, ya que con 32 bits se obtiene la resolución necesaria para resolver el problema y a su vez, se ocupa un menor espacio de almacenamiento.

La representación de datos en coma flotante de 32 bits sigue el estándar IEEE 754. Este estándar tiene varios campos que son el signo el cual ocupa un bit, el exponente que tiene un tamaño de 8 bits y la mantisa con un tamaño de 24 bits. En la figura 10, podemos encontrar una representación indicando el tamaño de bits que ocupa cada una de las partes que componen este estándar.

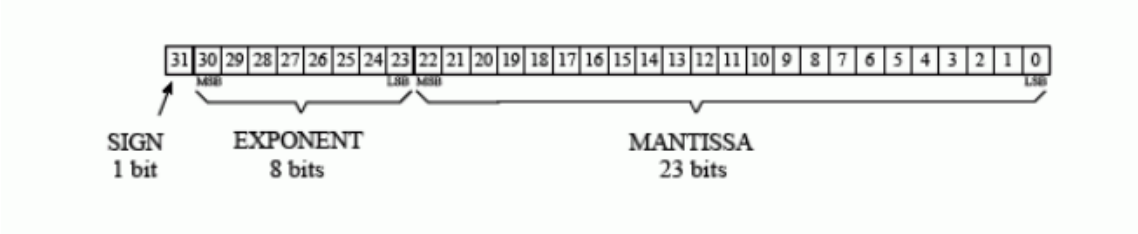


Figura 10: Representación estándar IEEE 754 [17].

Para representar un número decimal en el estándar IEEE 754, se deben seguir los siguientes pasos: Primero, se ha de convertir el número decimal a representación binaria; después, sería necesario convertir esta representación binaria en notación científica. Una se vez se hayan realizado los pasos anteriores, se pueden rellenar los 32 bit de la representación IEEE 754. El primer bit como se ha mencionado anteriormente corresponde con el signo, recibirá un valor de 1 si es negativo, 0 si es positivo. Los siguientes 8 bits, corresponden a la suma del exponente de la notación científica, más una constante la cual tiene un valor de 127. Por último, los 23 bits restantes pertenecen a la parte fraccionaria de la notación científica.

Una vez se haya realizado el entrenamiento en Matlab de forma satisfactoria, se obtienen los distintos pesos y bias, así como el gain de cada neurona de entrada. Al haber realizado el entrenamiento, la red ha aprendido de los distintos datos de entrada proporcionados durante el entrenamiento, ajustando los pesos y bias adecuados para una correcta ejecución y éstos serán los pesos y bias utilizados la implementación hardware.

Hay que tener en cuenta, que estos datos proporcionados por Matlab tienen una representación decimal, por lo que es necesario transformar todos los datos a coma flotante, debido a que son muchos datos para realizar esta conversión a mano. Se ha hecho uso de un código creado por usuarios de Matlab, el cual permite la conversión de decimal a IEEE 754 de 64 bits. Debido a que los datos tratados en la red neuronal creada tienen un tamaño de 32 bits, se han hecho diversas modificaciones en el código, para que los datos transformados en el estándar IEEE 754 tengan un tamaño de 32 bits y no 64. Esto se ha conseguido, delimitado el resultado obtenido a un tamaño de 32 bits, restructurando los bits ocupados por el exponente y la mantisa.

Para el correcto funcionamiento de la red, es necesario que los valores de los pesos y bias, estén guardados en una memoria, con la finalidad de que la red pueda acceder a esos valores cuando sea necesario. Para llevar a cabo esta tarea, se ha hecho uso de una memoria RAM, dentro de los bloques IP que integra Vivado, aparte de bloques para realizar operaciones, podemos encontrar muchos más como en este caso, un bloque llamado Distrubuted Memory Generator. Este bloque nos permite configurar la clase de memoria que el usuario necesite, ROM o RAM, con variedad de configuraciones adicionales, así como el tamaño de los datos guardados y el tamaño de la memoria. Obviamente, en este caso el tamaño de los datos almacenados en la memoria es de 32 bits, en cuanto al tamaño de la memoria dependerá de la finalidad de cada uno de los bloques de memorias necesarios a lo largo del proyecto.

Estos bloques de memoria, a su vez cuentan con la opción de llenar el contenido de la memoria con unos valores iniciales, mediante un archivo llamado COE file o coefficients file, el archivo .coe pasará al núcleo de la memoria como un archivo MIF Memory Initialization File, archivo de inicialización de la memoria. El archivo .coe tiene una estructura determinada, en la primera línea se debe definir la base en la que están representados los datos que se van a escribir en el archivo, en este caso, los datos utilizados estarán representados en hexadecimal, para una mayor simplicidad a la hora de representar estos datos. Después, ya se puede escribir el vector de datos con el que se desea rellenar la memoria, Cada dato deberá ir separado mediante comas y el último dato con un punto y coma, para indicar el final del vector. La figura 11, representa un ejemplo de un archivo .coe, donde podemos observar la estructura del archivo, en la primera línea se define la base en la que se van a representar los valores mediante memory_initialization_radix.

```
memory_initialization_radix=16;
memory_initialization_vector=
00000000,
00000000,
431F0000,
00000000,
00000000;
```

Figura 11: Ejemplo archivo .coe.

Teniendo en cuenta los conceptos anteriores, se procederá a explicar la solución propuesta para conseguir la implementación hardware. La idea básica de la implementación propuesta reside en varias memorias, las cuales tienen asociadas su controlador correspondiente, estos controladores se encargarán de mandar y recibir señales del resto de las memorias y de los bloques encargados de realizar las operaciones correspondientes formando una cadena.

El número de neuronas en cada capa se establece en cada controlador, mediante un número genérico llamado value. Esto permite al usuario la capacidad de cambiar el número de neuronas en cada capa según las necesidades de la red.

Se han utilizado varios bloques distintos de memoria RAM, permitiendo separar los distintos datos en bloques de memoria diferentes. Siendo necesarios un bloque de memoria para almacenar los resultados de cada capa, dando lugar a una memoria para la capa de entrada, otra memoria para la capa oculta y una última para la capa de salida. También serán necesarios bloques de memoria para guardar los pesos, bias y gain proporcionados por Matlab, y una memoria adicional para guardar los resultados de la función softmax, formando un total de 7 bloques de memoria.

A continuación, se explicará en detalle la función de cada una de las memorias escogidas:

Matlab nos proporcionará una matriz la cual contiene los diferentes pesos que componen la red, hay que tener cuidado, debido a que la matriz no contiene los 7840 pesos necesarios para ejecutar la red, si no que contiene 7817. Al entrenar la red obtenemos dos matrices distintas, una para los pesos de las conexiones entre la capa de entrada y la capa oculta, y otra matriz para los pesos correspondientes a las conexiones, entre la capa oculta y la capa de salida. La primera matriz tiene unas dimensiones de 717x10 y la segunda unas dimensiones de 10x10. Tal y como se ha explicado antes, cada neurona de entrada establece una conexión con cada neurona de la cada oculta, dado que tenemos 784 neuronas de entrada y 10 en la capa oculta, la matriz debería ser 784x10. Las columnas representan el número de neuronas de entrada y las filas el número de neuronas en la capa oculta. Por ejemplo, la posición en la matriz 2x4 corresponde al peso de la conexión entre la neurona 2 de la capa de entrada y la neurona 4 de la capa de salida.

Como podemos observar, la matriz que nos proporciona Matlab tiene únicamente 717 columnas, es decir faltan los pesos correspondientes a 67 neuronas de entrada, debido a

que cada neurona de entrada se conecta con 10 de la capa oculta, faltando 670 pesos. Esto es debido, a que las neuronas de entrada que no aportan ningún valor en la imagen, principalmente las neuronas que componen las esquinas o bordes de la imagen son neuronas cuyo valor es siempre 0. Dado que estás neuronas no aportan nunca ningún valor los pesos calculados por Matlab para de sus conexiones correspondientes, también tienen un valor igual 0. Al obtener la matriz de los pesos, Matlab directamente no rellena las filas de estas neuronas porque lo pasa por alto, pero si tiene en cuenta que su valor es 0 a la hora de ejecutar la red.

En nuestro caso, tenemos que rellenar los pesos de todas las conexiones, aunque su valor sea igual a 0, debido a que la red que se va a diseñar no tiene en cuenta lo que se ha mencionado anteriormente. Es importante destacar, que por defecto los datos con los que opera Matlab tienen una precisión doble, es decir tienen un tamaño de 64 bits.

Los pesos deben de ser pasados de decimal a coma flotante y una vez que se tenga la representación de coma flotante serán convertidos en hexadecimal y guardados en un fichero. Esto se ha resuelto mediante un bucle for, el cual leerá los datos de la matriz y realizará varias tareas. Lo primero será convertir el número que se encuentra en la matriz de double a single para que tengo un tamaño de 32 bits, después se convertirá al estándar IEEE 754, y se convertirá el valor de binario a hexadecimal. Por último, es necesario agrupar la palabra en hexadecimal con una string, debido a que al hacer la conversión el resultado se muestra en variable char, lo que da problema a la hora de pasarlo a un fichero, Por ello, es mejor agruparlo en una cadena de caracteres. El último paso sería guardar el resultado hexadecimal en un archivo .coe, separando cada dato mediante comas. Este proceso será necesario en todos los bloques de memoria inicializados mediante un coe file. En el anexo 1 podemos encontrar el código diseñado para la creación del archivo .coe.

Gracias a esto, podemos llenar la memoria con todos los pesos obtenidos en Matlab. En esta memoria no será necesario escribir nada debido a que únicamente debe contener los valores de los pesos. El bloque de memoria reservado para guardar los pesos de la red cuenta con un total de 8192 posiciones, debido a que debe el total de ranuras del bloque de memoria deben ser múltiplos de 16.

La forma en la que se han ordenado los pesos en la memoria es la siguiente, debido a que no podemos formar una matriz, se ha decidido seguir un orden para llenar las 7940 posiciones de memoria. En la figura 12, podemos encontrar un esquema en el que se indican el orden que siguen los pesos en las distintas posiciones de la memoria, donde W_1 corresponde al peso ubicado en la primera posición de la memoria, W_2 al segundo… Esta imagen nos permite observar que la primera posición de la memoria será ocupada por el peso de la conexión entre la primera neurona de la capa de entrada y la primera neurona de la capa oculta. La siguiente ranura, será para el peso correspondiente a la conexión entre la segunda neurona de entrada y la primera neurona de la capa oculta y así hasta recorrer todas las conexiones correspondientes a las neuronas de entrada con la primera neurona de la capa oculta. Una vez se haya terminado de recorrer la capa de entrada, se

colocarán los pesos de las conexiones correspondientes a la segunda neurona de la capa oculta y todas las de la capa de entrada, y así sucesivamente hasta haber rellenado la memoria con los 7940 pesos.

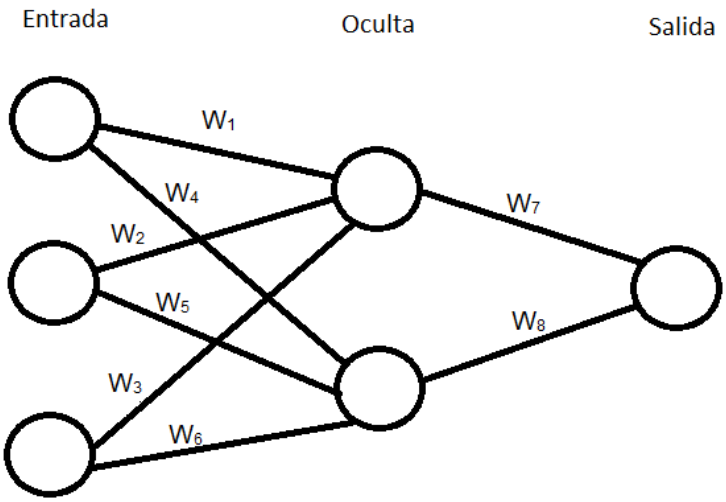


Figura 12: Esquema distribución de los pesos.

Respecto a la memoria encargada de guardar el valor de los bias, el llenado de la memoria también se hace mediante un archivo de coeficientes. En este caso, únicamente es necesario guardar 20 bias. Dado que tenemos un bias asociado a cada neurona de la capa oculta y la capa de salida, a la hora de almacenarlos en la memoria, el orden a seguir es el siguiente: Se guardarán en las primeras posiciones los bias correspondientes a las neuronas de la capa oculta, siguiendo el mismo orden que las neuronas de la capa. Después, una vez que se hayan colocado los 10 bias de la capa oculta, las siguientes 10 posiciones serán ocupadas para los 10 bias restantes de la capa de salida.

La siguiente memoria a rellenar mediante un archivo .coe, será la memoria encargada de almacenar el valor de los distintos gains. El gain es necesario para realizar el preprocesado de los datos de entrada que hace Matlab, por lo que tendremos un gain por cada dato de entrada, formando un total de 784 ganancias. La memoria encargada de almacenar estos datos tendrá un total de 1024 posiciones, el orden para almacenar los datos será el mismo orden que sigan las neuronas de entrada.

Por último, se encuentra el bloque de memoria encargado de almacenar los datos de entrada. El llenado de la memoria que almacena los datos de entrada se hará a través de un archivo .coe, al igual que en las situaciones anteriores. Hará falta una memoria con 1024 posiciones de profundidad para poder almacenar los 784 valores de entrada correctamente. El orden de los datos almacenados seguirá el orden de las neuronas de entrada, debido a que el dato almacenado es el valor de cada una de las neuronas de entrada y cada dirección de la memoria corresponderá a cada neurona de entrada.

El resto de las memorias utilizadas, servirán para almacenar los datos de la capa oculta, la capa de salida y los valores de la función softmax. Estas memorias se irán sobrescribiendo a medida que se ejecute la red, y no será necesario hacer una inicialización previa. El valor inicial de estas posiciones de memoria será igual a 0. Como se ha mencionado anteriormente, todas las memorias almacenarán datos con un tamaño 32 bit, expresados en bajo el estándar IEEE 754.

Todas las memorias tienen un controlador, el cual se encarga de diversas tareas, entre ellas incrementar las direcciones de la memoria mediante un contador incorporado en cada controlador. Todos los contadores se reiniciarán cuando alcancen su valor máximo, estipulado previamente. La entrada enable del controlador indica cuando se debe incrementar la dirección. La salida address tomará los valores del contador y será conectada con la entrada a de la memoria, encargada de representar la dirección. Una vez explicado la forma guardar los valores iniciales necesarios para resolver el problema, y las conexiones básicas entre los controladores y las memorias, se procederá a la explicación del funcionamiento de la red.

En la figura 13, podemos encontrar un diagrama de bloques, en el cual se representan los bloques más significativos que componen la red y sus conexiones.

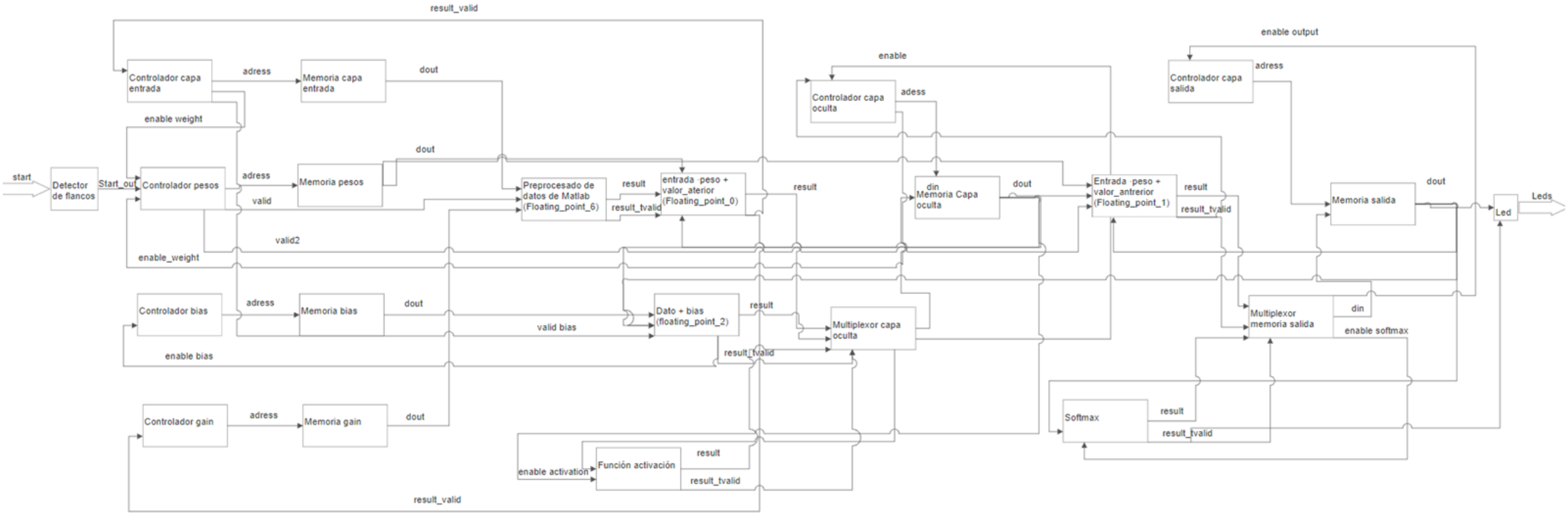


Figura 13 : Diagrama de bloques de la red neuronal.

Para comenzar la ejecución de la red, el usuario debe presionar el botón de inicio. En el programa recibe el nombre de start y es una entrada definida como std_logic. Esta entrada está conectada a un módulo llamado detector de flancos, la misión de este bloque es detectar cuando el usuario ha dejado de presionar el botón. Cuando esto ocurre se activa una señal llamada start_out, la cual se activará durante únicamente un ciclo de reloj, indicando a la red que se puede comenzar la ejecución. Es muy importante añadir este bloque, debido al ser una implementación hardware los procesos como operaciones, necesitan como referencia los ciclos del reloj, realizando una suma u operación por ciclo. Si la señal que les indica que pueden comenzar la operación está activa durante más de un ciclo de reloj, se realizará varias veces la misma operación. Teniendo en cuenta que un ciclo de reloj equivale a 10ns es necesario, añadir un detector de flancos en el botón dado que la pulsación del usuario durará varios ciclos de reloj.

La señal start_out, es recibida por el bloque controlador de la memoria encargada de almacenar los pesos, la misión de este controlador es: Recibir todas las señales relacionadas con los pesos y mandar las señales correspondientes para iniciar las operaciones cuando este todo correcto. A su vez, incluye un contador, el cual será el encargado de sumar las direcciones de la memoria a medida que avance el programa. En la figura 14, se muestra una imagen del bloque controlador de la memoria de los pesos. Mostrando las entradas y salidas de dicho módulo.

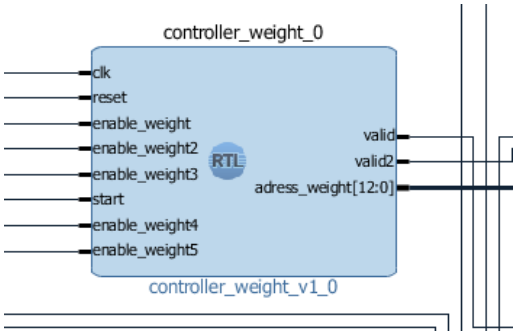


Figura 14: Controlador de la memoria encargada de almacenar los pesos.

El controlador de la memoria recibe esta señal como una entrada llamada start, cuando esta señal es igual a 1 significa que comienza la ejecución de toda la red. Todas las memorias se encuentran en la dirección 0, porque el programa acaba de empezar su funcionamiento. La memoria de la capa de entrada mostrará el valor de la primera neurona de entrada, así como la memoria encargada de almacenar el gain. La memoria de los pesos también mostrará el peso correspondiente a la primera operación. Por lo que directamente, se puede realizar la primera operación. Cuando la entrada start del controlador se active, este activará una señal de salida llamada valid, la cual durará un ciclo de reloj. En el anexo 4, podemos encontrar el código creado para el controlador de la memoria en la que se almacenan los pesos.

La señal valid, es recibida por el bloque floating_point_6, este bloque es una IP, que realizará la siguiente operación (A·B) + C, con la finalidad es realizar el preprocesado de datos de Matlab. Multiplicará el valor de las neuronas de entrada por el gain y restará 1. La entrada A de este módulo será la salida de la memoria encargada de guardar el gain, la entrada B del bloque corresponde con la salida de la memoria encargada de almacenar los datos de la capa de entrada. Por último, la entrada C corresponderá con una constante la cual tendrá un valor de -1, 0xbf800000 en representación correspondiente a coma flotante.

Todas estas entradas, son vectores std_logic de 32 bits, debido a que es el tipo de información que maneja la red. Por último, cada entrada (A, B, C) tendrá asociada una entrada llamada tvalid del tipo std_logic. Cuando esta entrada tenga un valor de 1, significa que el dato aportado, es el correcto para realizar la operación. Cuando las 3 entradas tvalid, tengan un valor igual a 1 se realizará la operación. Estás 3 entradas recibirán la señal valid del controlador correspondiente a la memoria de los pesos, por lo que las 3 tendrán siempre el mismo valor. En cuanto a las salidas del bloque, tenemos por un lado el resultado de la operación y por otro una salida llamada result_tvalid, std_logic. Esta salida, será igual a 1 cuando la operación haya finalizado, durante un ciclo de reloj. Ambas salidas estarán conectadas al bloque floating_point_0, encargado de realizar la siguiente operación.

Se puede realizar la operación en un ciclo de reloj, pero dado al tamaño de los datos y la gran cantidad de operaciones a realizar en este proyecto, no es posible realizar todas las operaciones garantizando un periodo de 100MHz en el reloj. Por lo que todos los bloques floating point utilizados están configurados con la máxima latencia, con una duración mayor a un ciclo de reloj.

El bloque floating_point_0, se encarga de realizar la misma operación que el bloque anterior (A·B) + C. Donde la entrada A, corresponde al resultado proporcionado por el bloque anterior (floating_point_6), correspondiente al preprocesado de los datos. La entrada B, corresponde a la salida de la memoria encargada de guardar los distintos pesos. La entrada C, corresponde a la salida de la memoria encargada de guardar los datos de las neuronas de la capa oculta. Como podemos observar, la finalidad de este bloque es obtener el resultado de las neuronas de la capa oculta. Multiplicando las neuronas de entrada por su el peso correspondiente a su conexión, y sumarle el valor almacenado en la neurona que se está intentando calcular. Por último, las 3 entradas tvalid, serán conectadas con la señal result_tvaid del floating_point_6, de modo que cuando la operación del preprocesado de datos haya sido realizada, se iniciará la siguiente operación, de forma encadenada. Como salidas de este bloque, obtendremos el resultado de la operación y una salida result_tvalid, que será igual a uno cuando se haya realizado la operación. Ambas salidas serán conectadas al bloque llamado multiplexor_memory_hidden.

A su vez, la salida result_tvalid está conectada a la entrada enable de los controladores de la memoria de la capa de entrada y la memoria del gain. Cuando la entrada enable sea

igual a uno y haya un flanco de subida de reloj, se incrementará el contador de los controladores, incrementando una posición las direcciones de las memorias. A su vez, cuando enable sea igual a 1, en el controlador de la memoria de entrada se activará la salida enable_weight. Esta salida estará conectada con el enable_weight del controlador de la memoria de los pesos, de modo que se incrementará la dirección de memoria, seleccionando en siguiente peso. El controlador de la memoria encargada de almacenar los pesos posee varios enable, debido a que la dirección de esta memoria se debe incrementar en distintas ocasiones. En la figura 15, se muestran las entradas y salidas del bloque controlador de la memoria de capa de entrada.

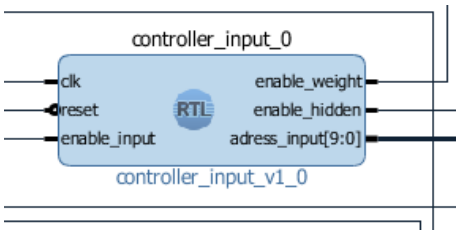


Figura 15: Controlador de la memoria de la capa de entrada.

En la memoria de la capa oculta se almacenarán los valores de las neuronas de la capa oculta, reservando una dirección de memoria para cada neurona. Los valores de las neuronas se irán actualizando a media que se ejecute la red, en diversas ocasione; por lo cual, la memoria recibirá datos de entrada y habilitaciones de escritura de diferentes bloques y por consiguiente hay que crear un multiplexor para escoger que datos se deben escribir en la memoria y la señal de escritura que se debe utilizar en cada momento.

El bloque multiplexor_memory_hidden cuenta con varias entradas std_logic_vector de 32 bits, para los datos que se deben guardar en la memoria de la capa oculta. Una de ellas, es la entrada din, que estará conectada a la salida del bloque anterior, floating_point_0, encargado de realizar el sumatorio para calcular el valor de las neuronas de la capa oculta. Otra entrada que podemos encontrar en este módulo es el we_normal que está conectado a la salida result_tvalid del anterior bloque. De modo que cuando esta entrada se active, significará que se realizado correctamente la operación anterior y qué se puede guardar en la memoria de la capa oculta. Como salidas, podemos encontrar dout, encargada de representar el dato que se almacenará en la memoria y we, la señal de escritura de la memoria. Cuando we_normal sea igual a 1, la salida dout tomará los valores de la entrada din y la salida we tomará los valores de we_normal. En la figura 16, se presenta el bloque multiplexor_memory_hidden con las entradas y salidas correspondientes.

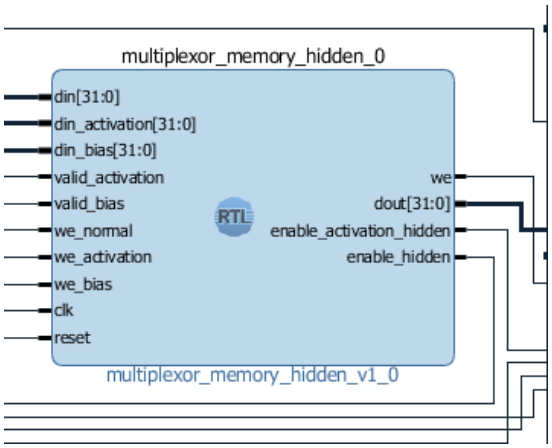


Figura 16: Multiplexor_memory_hidden.

Ambas salidas serán conectadas a la memoria encargada de almacenar los valores de la capa oculta. A su vez, posee una salida enable_hidden que será conectada al controlador de la memoria de la capa oculta. En el anexo 7, podemos encontrar el código utilizado para diseñar este módulo, podemos visualizar que este módulo tiene más entradas y salidas que las mencionadas, pero no serán explicadas hasta que se haga uso de ellas, con el fin de no complicar la compresión de la solución propuesta.

En la memoria encargada de almacenar los valores de la capa oculta, la entrada d de la memoria, recibirá la salida dout del multiplexor_memory_hidden y la entrada we, el valor de la salida we del multiplexor_memory_hidden, que será la confirmación de escritura. Cuando la entrada we de la memoria sea igual a 1 y haya un flanco de subida de reloj, se almacenará en la memoria el valor de la entrada d, en la dirección definida por la entrada a. Esta entrada a se encuentra conectada al controlador de la memoria de la capa oculta.

El controlador de la capa oculta se encarga de recibir las señales necesarias para el correcto funcionamiento de la red, además funciona como un contador, incrementando la dirección de la memoria de la capa oculta. La salida del controlador llamada adress_hidden tomará los valores del contador y será conectada a la entrada a de la memoria, que representa la dirección de dicha memoria. En la figura 17, podemos encontrar una imagen del bloque encargado de controlar la memoria de la capa oculta.

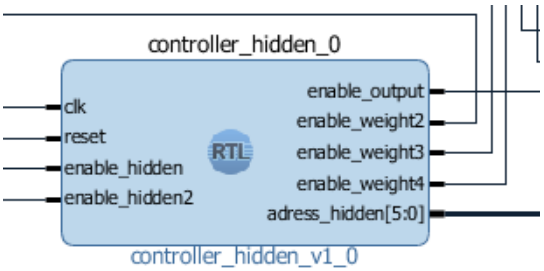


Figura 17: Controlador memoria capa oculta.

Llegados a este punto hemos conseguido almacenar en la primera neurona de la capa oculta, la operación realizada con la primera neurona de entrada y su peso correspondiente. Se han incrementado los contadores de las memorias de la capa de entrada, del gain y de los pesos, por lo que el sistema estará listo para realizar el mismo proceso, pero con la segunda neurona de la capa de entrada. Debido al paralelismo que presentan estos dispositivos, este proceso ya estaba teniendo lugar mientras se guardaba el resultado en la memoria de la capa oculta.

En el controlador de los pesos, cuando enable_weight es igual a 1 y hay un flanco de subida de reloj, se incrementa la dirección de la memoria como se ha había mencionado anteriormente. Así mismo, la salida valid de este módulo es igual a 1 durante un ciclo de reloj. Tal y como se ha explicado anteriormente, esta salida se encuentra conectada a las 3 entradas tvalid que podemos encontrar en el floating_point_6. Cuando estás entradas sean igual a 1 se realizará de nuevo el preprocesado de los datos de entrada y todos los demás procesos explicados anteriormente. Realizando el sumatorio de los pesos por las distintas neuronas de entrada, más el valor almacenado anteriormente en la neurona de la capa oculta. Esto proceso tiene lugar 784 veces hasta haber recorrido todas las neuronas de la capa de entrada.

Cuando se haya realizado la operación correspondiente a la última neurona de entrada, y el enable de controlador de la memoria de la capa de entrada sea igual a 1. El contador al haber llegado a 784 se reiniciará y será igual a 0, por lo que la memoria de la capa de entrada mostrará el valor de la primera neurona de entrada. Cuando esto tenga lugar, la señal de salida del controlador enable_hidden será igual a 1. Esta salida, está conectada con la entrada del multiplexor_1. El bias se debe sumar una vez se haya terminado el cálculo de una neurona de la capa oculta o la capa de salida. Para ahorrar recursos, los mismos módulos se encargarán de sumar ese bias tanto a las neuronas de la capa oculta, como a las neuronas de la capa de salida. Por eso, es necesario multiplexar la entrada de los bloques correspondiente a realizar la operación para saber a qué capa pertenece el dato que se está tratando, el multiplexor_1 se encargará de llevar a cabo esta tarea.

En el multiplexor_1, dispone de varias entradas las cuales son: din_hidden, esta entrada está conectada a la salida de la memoria de la capa oculta. Din_output, esta entrada irá conectada a la salida de la memoria de la capa de salida. Enable_hidden, que será igual a 1 cuando el dato proporcionado sea de la capa oculta. Enable_output, esta señal será igual a 1, cuando el resultado provenga de la capa de salida.

Como salidas podemos encontrar: dout, que corresponde al dato seleccionado y valid_bias. A su vez, este módulo cuenta con una salida llamada identifier, que tomará un valor igual a 0, cuando enable_hidden sea 1 y un valor igual a 1, cuando enable_output sea igual a 1. Este identifier estará conectado al módulomultiplexor_out_1, el cual será explicado más adelante. Cuando enable_hidden sea igual a 1, dout tomará los valores de la entrada din_hidden y valid_bias será igual a 1 durante un ciclo de reloj. Este valid_bias, estará conectado a las entradas tvalid del floating_point_2,

El bloque floating_point_2, será el encargado de sumar el bias a la neurona de la capa oculta, realizará la siguiente operación A+B. Donde A, será la salida dout del multiplexor_1, es decir el valor de la neurona de la capa oculta. Mientras que B, corresponde a la salida de la memoria en la cual se almacenan los bias. Cuando tvalid, sea igual a 1, se realizará la operación. La salida correspondiente al resultado de la operación será conectada a la entrada din_bias del bloque multiplexor_memory_hidden. La salida result_tvalid estará a la entrada valid de la entidad multiplexor_out_1.

El módulo multiplexor_out_1, tiene dos entradas, valid que será igual a 1, cuando se haya calculado el bias y la entrada identifier, la cual está conectada con la salida identifier del bloque multiplexor_1. Si identifier es igual a 0, signifca que el dato proviene de la capa oculta, cuando tiene un valor igual a 1, significa que el bias calculado pertence a la capa de salida. Como salidas de este bloque, encontramos valid_hidden y valid_output. Valid_hidden estará conectada a las entradas we_bias y valid_bias del bloque multiplexor_memory_hidden. Si la entrada valid es igual a 1 e identifier es igual a 0, la salida valid_hidden será igual a 1. Si, por el contrario, valid es igual a 1 e identifier es igual a 1, la salida valid_output será igual a 1.

Si la entrada valid_bias, del bloque multiplexor_memory_hidden es igual a 1, la salida dout del bloque tomará los valores de la entrada din_bias y la salida we, tomará los valores de la entrada we_bias. We_bias también será igual a 1, porque posee el mismo valor que valid_bias, almacenando el resultado de la suma del bias en la memoria de la capa oculta. Cuando la entrada valid_bias sea igual a 1 y haya un flanco de subida del reloj, la salida enable_activation_hidden de este bloque será igual a 1. Esta salida se encuentra conectada al bloque floating_point_8, con el fin de realizar el cálculo de función de activación.

La función de activación a realizar es la tangente hiperbólica, cuya expresión es $\tanh = \frac{2}{1+e^{-2\cdot x}} - 1$. Esta ecuación, deberá ser realizada por partes mediante varios bloques floating_point. El bloque floating_point_8, será el encargado de realizar la operación A·B, que representa a la operación -2·x de la ecuación anterior. Donde la entrada A, es la salida de la memoria de la capa oculta. Y la entrada B, será una constante la cual tendrá un valor de -2.

La entrada tvalid del floating_point será conectada a la salida enable_activation_hidden, del bloque multiplexor_memory_hidden, de modo que cuando enable_activation_hidden tenga un valor igual a 1, comenzará la operación. El resultado de la operación será conectado al bloque floating_point__3, así como el result_tvalid, indicador de que sea realizado la operación de forma satisfactoria. Estas conexiones se harán de la misma manera en todos los bloques floating_point sucesivos, encargados de hacer el cálculo de la función de activación.

El bloque floating_point__3, llevará a cabo la siguiente operación e^A. Donde A, será el resultado de la operación realizada por el floating_point__8 (-2·x). El resultado y el result_tvalid correspondiente, serán conectados al siguiente bloque, el módulo floating_point__4.

El siguiente bloque es el módulo floating_point__4, realizará la operación A+B, equivalente a $1+ e^{-2 \cdot x}$ de la función de activación. Donde A, será conectado a una constante con un valor 1 y la entrada B, será conectada al resultado de la operación anterior. La salida, resultado de esta operación y su result_tvalid serán conectados al bloque floating_point__5.

El módulo floating_point__5, realizará la operación A/B. Donde la entrada A, será conectada a una constante, la cual tendrá un valor igual a 2. La entrada B será conectada al resultado de la operación anterior realizada por el bloque floating_point__4. La salida, resultado de esta operación y su result_tvalid serán conectados al módulo floating_point__9.

Por último, el bloque floating_point__9, será el encargado de realizar la operación A-B. Donde la entrada A será el resultado del bloque anterior floating_point y la entrada B una constante la cual tendrá un valor igual a 1. La salida de este bloque resultado de la operación y su result_tvalid serán conectados al módulo multiplexor_memory_hidden.

La entrada din_activation del módulo multiplexor_memory_hidden, será conectada al resultado de la operación anterior, siendo el resultado final de la función de activación. Las entradas we_activation y valid_activation serán conectadas a la salida result_tvalid del bloque floating_point__9. Cuando valid_activation sea igual a 1, significará que se ha realizado con éxito el cálculo de la operación anterior, por lo que salida we del bloque multiplexor_memory_hidden, tomará los valores de la entrada we_activation. La salida dout del mismo bloque tomará los valores de din_activation. Como ya sabemos, ambas salidas irán conectadas a la memoria de la capa oculta, de modo que cuando we sea igual a 1, se escribirá en la memoria el dato encontrado en dout.

Llegados a este punto se habría conseguido realizar el cálculo completo de la primera neurona de la capa oculta, por lo cual, ya se podría pasar a realizar el cálculo de la segunda neurona de la capa oculta. Para ello, cuando la entrada valid_activation sea igual a 1 y haya un flanco de subida del reloj, la salida enable_hidden del bloque multiplexor_memory_hidden, será igual a 1. Esta salida será conectada a la entrada enable_hidden del controlador de la memoria de la capa oculta.

Si la entrada enable_hidden del controlador de la memoria de la capa oculta, es igual a 1, se incrementará el contador de dicho controlador, por lo que se incrementará la dirección de la memoria de la capa oculta, en esta dirección se almacenará el dato de la segunda neurona de la capa de salida. Cuando esto tenga lugar, la salida enable_weight4 de este bloque será igual a 1. Esta salida estará conectada al controlador de la memoria encargada de almacenar los pesos.

La memoria de la capa de entrada y la memoria encargada de almacenar el gain ya se encuentran en la dirección 0 porque ya se han reseteado los contadores al llegar a su valor máximo. Sólo faltaría incrementar la dirección de la memoria encargada de almacenar los pesos, para poder comenzar el cálculo de la siguiente neurona.

Si la entrada enable_weight4 del controlador de la memoria de los pesos sea igual a 1, se incrementará una posición de esta memoria, seleccionando el siguiente peso. Cuando esto tenga lugar, la salida valid de este controlador será igual a 1. Como ya se ha explicado anteriormente, la salida valid, se encuentra conectada a la entrada t_valid del floating point_6 encargado de hacer el preprocesado de los datos.

Cuando la entrada tvalid sea igual a 1 se realizará el mismo proceso anterior. Se hará un preprocesado de los datos, se multiplicarán por un peso y se le sumará el resultado encontrado en la segunda neurona de la capa oculta. Después, se almacenará ese resultado sobrescribiendo el valor de la segunda neurona de la capa oculta. Esto se realizará hasta haber recorrido todas las neuronas de la capa de entrada. Una vez recorridas, se le sumará el bias correspondiente y se hará el cálculo la función de activación. Este resultado será el resultado de la segunda capa oculta, que será almacenado en la memoria de la capa oculta. Llegados a este punto se puede pasar a realizar el cálculo de la siguiente neurona de la capa oculta realizando el mismo proceso. Esto deberá hacerse un total de 10, debido a que tenemos 10 neuronas en la capa oculta.

En el momento en que el controlador de la capa oculta haya contado hasta 10, significará que se han calculado el valor de todas las neuronas de la capa oculta. Entonces se reiniciará el contador seleccionando la posición 0 que corresponde a la primera neurona de la capa oculta. A la vez que ocurre esto, se activará la salida enable_weight2, del controlador de la memoria de la capa oculta. Esta salida será conectada al controlador de la memoria encargada de almacenar los pesos. Si la entrada enable_weight2 del controlador correspondiente a la memoria de los pesos, es igual a 1 se activará la salida valid2. Esta salida estará conectada al módulo floating_point_1.

El bloque floating_point_1, será el encargado de realizar el cálculo de las neuronas de la capa de salida. Este bloque realizará la operación (A·B) + C, es decir multiplicará las neuronas de la capa oculta por un peso y sumará el resultado encontrado en la neurona de salida que se pretende calcular. Donde la entrada A, será la salida de la memoria de la capa oculta. La entrada B, estará conectada a la salida de la memoria de los pesos. Por último, la entrada C será conectada a la salida de la memoria de la capa de salida. Las tres entradas tvalid, irán conectadas a la salida valid2, del controlador de la memoria de los pesos. El resultado de la operación y su result_tvalid irán conectados al módulo multiplexor_memory_out.

En la memoria de la capa de salida se guardarán datos procedentes de varias entradas, por lo que es necesario multiplexar esos datos, al igual que ocurre con la memoria de la capa oculta.

El módulo multiplexor_memory_out, funciona de la misma forma que el módulo multiplexor_memory_hidden. Podemos encontrar la entrada din que irá conectada al resultado de la operación anterior, realizada por el bloque floating_point_1. Y la entrada we_normal, la cual será conectada a la salida result_tvalid del bloque floating_point_1. Por defecto la salida we, siempre tomará los valores de we_normal, al igual que la salida

dout, tomará los valores de din. De este modo, cuando we sea igual a 1, se escribirá en la memoria de la capa de salida el valor encontrado en dout. La figura 18, se presenta el bloque multiplexor_memory_out con las entradas y salidas correspondientes.

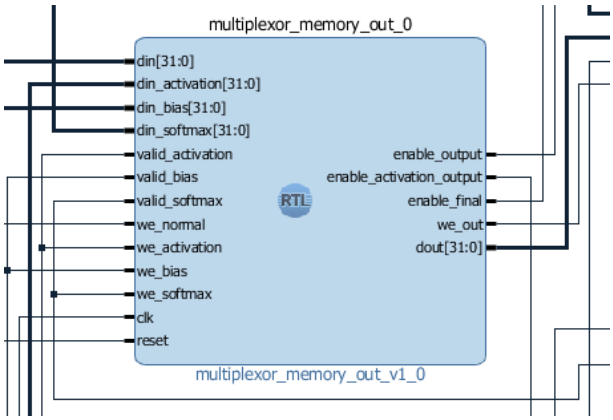


Figura 18: Multiplexor_memory_out.

Por otro lado, la salida result_tvalid del bloque floating_point_1, encargado de realizar el cálculo de las neuronas de la capa de salida. Se encuentra conectada a la entrada enable_hidden del controlador de la memoria de la capa oculta. Cuando se haya realizado la operación de forma correcta, se incrementará el contador. Incrementando la dirección de la memoria de la capa oculta se podrá continuar con el cálculo de la primera neurona de la capa de salida. A su vez cuando esto ocurra, la salida enable_weight3 de este bloque se activará, esta salida estará conectada a la entrada enable_weight3 del controlador de la memoria de los pesos, aumentando la dirección y seleccionando el siguiente peso.

Cuando se haya incrementado la dirección de la memoria de los pesos, el controlador de dicha memoria activará la salida valid2 durante un ciclo de reloj. Como ya sabemos esta salida irá conectada al floating_point_1 y se repetirá todo el proceso mencionado anteriormente. Este proceso, tendrá lugar 10 veces, hasta que se hayan recorrido todas las neuronas de la capa oculta. En este instante, se debería sumar el bias correspondiente a la primera neurona de salida y realizar la función de activación. Una vez, el controlador de la memoria de la capa oculta llegue a 10, se resteará y tomará un valor igual a 0. La salida enable_output del controlador será igual a 1. Esta salida irá conectada al multiplexor_1.

El multiplexor_1, es el encargado de multiplexar la entrada del bloque floating_point_2, encargado de sumar el bias. La entrada din_output estará conectada a la memoria de salida y la entrada enable_output estará conectado al controlador de la memoria de la capa oculta. Cuando la entrada enable_output sea igual a 1, la salida identifier tendrá un valor de 1 indicando que el valor procede de la neurona de la capa de salida. El dout tomará los valores de din_output y valid_bias será igual a 1, durante un ciclo de reloj. Estas dos últimas salidas serán conectadas al floating_point_2.

La salida dout del bloque floating point_2, será conectada al bloque multiplexor_memory_out, encargado de multiplexar la memoria de salida. Cuando la operación sea la correcta, la salida result_tvalid será uno. Esta salida estará conectada al bloque multiplexor_outuput. Al igual que la salida identifier del bloque multiplexor_1, que será conectada a la entrada identifier del bloque multiplexor_output. Si la salida indenfier es igual a 1, significa que el bias calculado pertenece a la neurona de salida, por lo que valid_output será igual a 1. La salida valid_output será conectada a las entradas we_bias y valid_bias del módulo multiplexor_memory_output.

Si la entrada valid_bias del bloque multiplexor_memory_out es igual a 1, la salida dout del bloque tomará los valores de la entrada din_bias y la salida we tomará los valores de la entrada we_bias. Esto permitirá que se sobrescriba en la memoria el valor tras haber sumado el bias. A su vez, si hay un flanco de subida del reloj y valid_bias es igual a 1, la salida enable_activation_output será igual a 1, esta salida estará conectada a la entrada tvalid del bloque floating_point_7.

El bloque floating_point_7, será el encargado de comenzar el cálculo de la función de activación, realizará la operación e^A. Donde la entrada A, estará conectada a la salida de la memoria de la capa de salida. El resultado de la operación será conectado a la entrada de din_acitvation del módulo multiplexor_memory_out, y la salida result_tvalid será conectadas a las entradas we_activation y valid_activation de ese mismo bloque. Con la finalidad de guardar los valores en la neurona de salida. A su vez, el resultado de la operación será conectado al siguiente bloque el floating point_10, así como la salida result_tvalid, que será conectada a la entrada tvalid del bloque floating_point_10.

La función de activación que se debe realizar en las neuronas de la capa de salida es la función sofmax, la cual viene representada por la siguiente ecuación $Softmax = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$. Como podemos observar para llevar a cabo dicha ecuación primero es necesario calcular la función exponencial de todas las neuronas de salida. Por lo que se va a ir calculando la función exponencial de cada neurona de salida y se guardará la suma de todas en una memoria. La cual no tendrá ningún controlador asociado debido a que no es necesario cambiar de dirección ni mandar ninguna señal, solamente debe almacenar todos los valores de la suma en una misma dirección. La suma de las exponenciales será llevada a cabo en el bloque floating_point_10.

El bloque floating_point_10, realiza la operación A+B. Donde la entrada A, es el resultado de la operación anterior realizada por el floating_point_7. La entrada B es la salida de la memoria encargada de almacenar la suma de los valores de las exponenciales. La salida que contiene el resultado de esta operación será conectada a la entrada d de la memoria encargada de guardar los valores de las exponenciales. El result_tvalid, encargado de indicar que ha finalizado la operación, será conectado a la entrada we de la memoria, encargada de indicar la confirmación de escritura.

Como se ha explicado anteriormente, el resultado de la exponencial y la salida result_tvalid del bloque floating_point_7, estarán conectadas al módulo multiplexor_memory_out. Sobrescribiendo con el resultado de la exponencial cada neurona de salida. Al mismo tiempo, cuando hay un flanco de subida del reloj y la entrada valid_activation es igual a 1, la salida enable_output es igual a 1. Esta salida se conectará a la entrada enable_weight5, del controlador de la memoria de los pesos. De modo que cuando sea igual a 1, se incrementará una posición en la dirección de la dicha memoria, seleccionado el siguiente peso. También, la salida enable_output estará conectada al controlador de la memoria de la capa de salida.

La memoria encargada de almacenar los valores de las neuronas de la capa de salida tiene un controlador asociado, el cual tiene como entrada enable_output, que está conectada a la salida enable_output del bloque multiplexor_memory_out. Esta entrada representa el enable del contador, por lo que cuando es igual a 1, se incrementará una posición en la dirección de la memoria, dando lugar a la siguiente neurona de salida. En la figura 19, podemos encontrar el bloque encargado de controlar la memoria de la capa de salida.

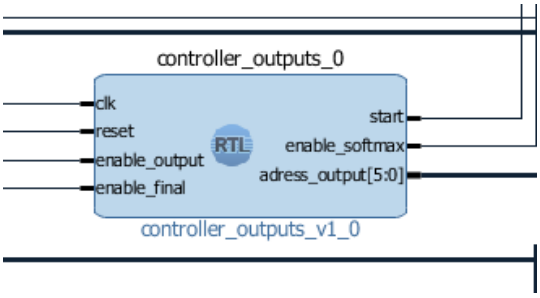


Figura 19: Controlador memoria capa de salida.

Por otro lado, cuando se incrementa el controlador de la memoria de los pesos mediante la entrada enable_weight5, se activa la salida valid2 de dicho controlador. Tal y como se ha explicado anteriormente esta salida valid2, se encuentra conectada a las entradas tvalid del bloque floating_point_1. Éste es el encargado de realizar el cálculo de las neuronas de salida, multiplicando las neuronas de la capa oculta por un peso, y sumar el valor guardado en la neurona que se está calculando. Se repetirá todo el proceso comentado anteriormente, calculando el valor de todas las neuronas de salida. Cuando se haya calculado el valor de las 10 neuronas de la capa de salida, sólo faltaría realizar el último paso de la softmax, dividiendo el resultado de cada exponencial entre la suma total.

Cuando el controlador de la memoria de la capa de salida haya contado 10 posiciones, significará que se ha realizado el cálculo de todas funciones exponenciales, por lo que se activará la salida start del controlador. Esta salida será conectada a la entrada start del bloque mux_final.

El bloque encargado de realizar la operación tendrá dos señales distintas, que le indicaran cuando se debe comenzar a la operación. Una de ellas es la entrada start, mencionada anteriormente y la otra es la entrada enable, que será explicada más adelante. Cuando la entrada start tenga un valor igual a 1, la salida enable_softmax será igual a 1. Esta salida se encuentra conectada a las entradas tvalid del bloque floating_point_11.

El bloque floating_point_11, realizará la operación A/B. Será el encargado de dividir cada exponencial que se encuentra almacenada en las neuronas de salidas, entre la suma total almacenada en una memoria. Donde A, será la entrada la salida de la memoria de la capa de entrada. Y la entrada B, será la salida de la memoria encargada de almacenar la suma total de todas las exponenciales. El resultado de la operación será conectado a la entrada din_softmax del bloque multiplexor_memory_out. La salida result_tvalid, será conectada a las entradas we_softmax y valid_softmax del mismo bloque.

El resultado de esta última operación será el valor definitivo de la neurona de salida. Si la entrada valid_softmax del bloque multiplexor_memory_out, es igual a 1, la salida dout, tomará los valores de la entrada din_softmax resultado de la operación. Y la salida we tomará los valores de la entrada we_softmax. De este modo, se guardará en la memoria de la capa de salida los valores obtenidos como resultado de esta operación.

Como ya sabemos, cada neurona de salida representará a un número del 0 al 9 y al final de la ejecución de la red cada neurona tendrá un valor del 0 al 1. Este valor nos indicará las probabilidades que tiene cada neurona de ser el resultado de la operación. Es decir, si la neurona 2 tiene un valor de 0.9, significa que existen un 90% de probabilidades de que el número representado en la imagen sea un 2. Debido a esto, la salida result_tvalid del floating_point_11, encargado de realizar la operación, será conectada también a las entradas tvalid del floating_point_12.

El bloque floating_point_12 es un comparador, que nos permitirá saber si la neurona de salida es mayor que un valor previamente estipulado. Este bloque comparará si A>B, y si esto ocurre, el resultado obtenido a la salida del bloque será igual a 1, sino será igual a 0. La entrada A de este módulo, estará conectada a la salida de la memoria de la capa de salida. La entrada B, es conectada a una constante la cual tendrá un valor de 0.6. así se comprueba si el valor final de la neurona calculada es mayor que 0.6; es decir, si tiene más de un 60% de posibilidades de ser el dígito representando en la imagen. La salida será el resultado de esta operación, y estará conectada a la entrada llamada din del bloque led. La salida result_tvalid será conectada a la entrada valid del bloque led.

Este bloque led será el encargado de recoger el resultado de la comparación y almacenar su valor en una señal interna de 10 bits, llamada data. Este vector irá del 9 al 0, donde cada bit representará a cada neurona de salida. Al comienzo, todos los bits de la señal data, son 0. Según se vayan calculando los valores de salida, se irá actualizando el bit correspondiente a cada neurona, con la ayuda de un contador. Escribiendo un 0 si son menores de 0.6 y un 1 si son mayores.

Por ejemplo, si se acaba de calcular el valor de la neurona 2 y es mayor a 0.6, la entrada din del bloque será igual a 1. El contador, en ese caso tendrá un valor de 2, por lo que en la posición 2 de la señal data escribirá un 1. Así se irán rellanando los 10 bits de la señal data a medida que se complete el cálculo de las neuronas de salida.

Mientras que tienen lugar todos los pasos descritos anteriormente, en el bloque multiplexor_memory_out se comprueba si hay un flanco de subida del reloj, y si la entrada valid_softmax del bloque multiplexor_memory_out es igual a 1. Si es así, la salida enable_final será igual a 1, y estará conectada a la entrada enable_final del controlador de la memoria de la capa de salida.

Si la entrada enable_final del controlador de la memoria de la capa de salida es igual a 1, se incrementará el contador, aumentado la dirección memoria de la capa de salida, seleccionado la siguiente neurona. En esta neurona encontramos el resultado de la función exponencial, por lo que sería necesario dividir ese valor entre la suma de todas las exponenciales, tal y como se ha realizado anteriormente. Para ello cuando se incremente una posición en el contador, la salida enable_softmax será igual a 1, esta salida se encuentra conectada a la entrada enable del bloque mux_final.

El módulo mux_final será el encargado de multiplexar la señal que habilita la operación, encargada de dividir el valor de cada exponencial, entre la suma total. Cuando la entrada enable de este bloque sea igual a 1, la salida enable_softmax de este bloque será igual a 1.

Como ya sabemos esta salida estará conectada al floating_point_11, encargado de realizar la división. Esto se debe realizar 10 veces, hasta que se haya conseguido calcular el valor final de todas las neuronas de la capa de salida. Cuando esto ocurra, todos los contadores que se encuentran en los controladores de las memorias habrán llegado a su valor máximo, por lo que se reiniciarán seleccionando la posición 0 de cada memoria.

Por otro lado, el contador ubicado en el bloque led, también habrá alcanzado su máximo valor, reiniciándose. Cuando esto ocurra, significa que la señal interna data habrá actualizado su valor escribiendo un 1 en la posición correspondiente a la neurona de salida que tenga un valor mayor que 0.6, debido a que ese será el dígito reconocido por la red. Cuando el contador de este bloque llegue a su máximo valor, la señal interna ok, tendrá un valor igual 1. Si la señal interna ok, es igual a 1, un case comprobará el valor del vector data. Dependiendo del valor del vector data las salidas led0, led1, led2 y led3 tendrán un valor igual a 1 o igual a 0. Estas salidas serán las encargadas de representar en binario el número que se encuentra en la imagen mediante los leds de la FPGA.

Retomando el ejemplo anterior, si el vector data muestra el siguiente resultado “0000000100”, quiere decir la posición 2 es mayor que 0.6, debido a que las posiciones van del 9 al 0. Esta posición equivaldrá a la segunda neurona de salida, encargada de representar el dígito 2 (suponiendo que se empieza en la neurona 0). Lo que significa que la imagen introducida en la entrada tiene más de un 60% de posibilidades de representar el número 2. Por lo que, la salida led3 será igual a 0, la salida led2 será igual a 0, la salida

led1 será igual a 1 y la salida led0 será igual a 0. Formando en binario la palabra “0010”, cuya representación decimal es 2. En el caso de que el número representado en la imagen corresponda al número 0, se encenderán todos los leds. Aunque esta no sea la representación binaria del 0, se ha decido encender todos los leds, para poder diferenciar correctamente de cuando el programa no esté funcionando. Ya que cuando no se ejecuta la red, todos los leds permanecen apagados. En la figura 20, podemos ver una fotografía de este ejemplo, en la que los leds están ubicados dentro del rectángulo rojo.

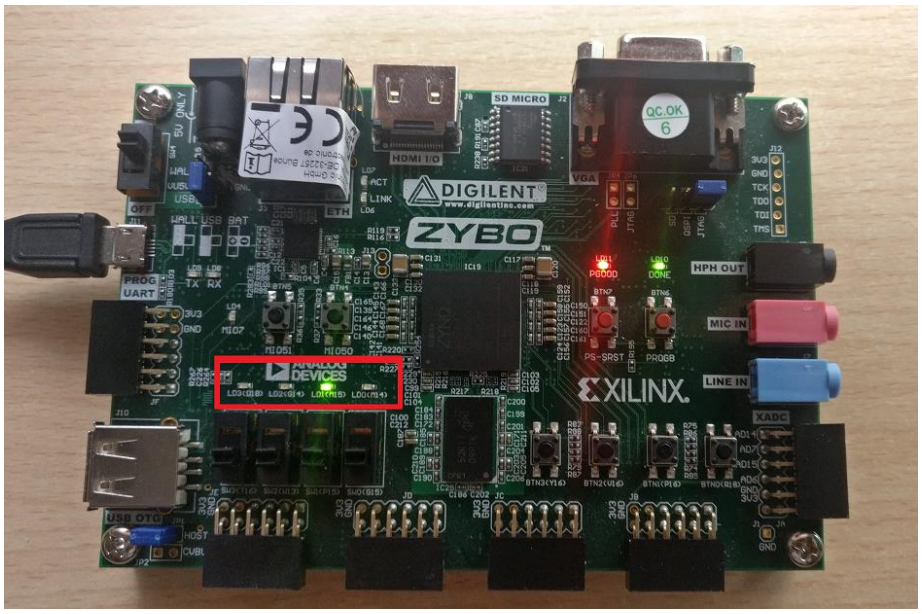


Figura 20: Representación del número reconocido.

Las salidas del sistema serán las led3, led2, led1 y led0, las cuales mediante un archivo XDC serán asociadas a los pins M14, M15, G14, D18 que corresponden a los 4 leds encontrados en la placa.

3.4. Mejoras en la arquitectura

Como se ha mencionado en el objetivo, una vez alcanzado la solución al problema, se han planteado varias mejoras, las cuales son reducir el tiempo de ejecución y realizar un proyecto con tablas de aproximación.

3.4.1. Reducción del tiempo de ejecución

El tiempo de ejecución de la red viene dado por los ciclos de reloj necesarios para alcanzar el resultado final y que se enciendan los leds. Donde cada ciclo de reloj equivale a 10ns, debido a que el reloj está programado con una frecuencia de 100MHz. Para el diseño de la red, se ha intentado aprovechar la flexibilidad que ofrece una FPGA realizando varias tareas en paralelo e intentando reducir el tiempo de ejecución. Una vez se ha realizado con éxito la primera implementación, se descubre que el preprocesado de datos que realiza Matlab se hace más veces de las necesarias.

Las neuronas de entrada de la red toman los valores de cada píxel que componen la imagen en escala de grises. Estos valores deben de ser tratados antes de ser multiplicados por su peso correspondiente y así poder calcular el valor de las neuronas de la capa oculta. En la versión del programa explicada anteriormente, el preprocesado de datos se realizaba cada vez que era necesario calcular el valor de una neurona de la capa oculta, ejecutando la operación un total de 10 veces por cada neurona de entrada. Así llegamos a la conclusión, de que esta operación se realiza más veces de las que serían necesarias.

Si la primera vez que se realiza la operación, se guardase el valor obtenido en una memoria, nos evitaríamos tener que realizar el cálculo las 9 veces restantes. Dado que es necesario realizar esta operación en cada neurona de entrada para calcular una neurona de la capa oculta, debido a que nuestra red cuenta con 784 neuronas de entrada y 10 neuronas en la capa oculta. Esta operación se realiza 7840 veces, teniendo en cuenta que, para ejecutar cada operación, es necesario 16 ciclos del reloj (debido a que posee una latencia de 16), llegando a un total de 125440 ciclos de reloj. Si se consigue realizar una única vez esta operación y después guardar sus resultados, serían un total de 16 ciclos por cada operación, más un ciclo para guardar los resultados. Siendo un total de 13328 ciclos de reloj. A su vez sería necesario multiplexar la entrada del floating_point_0 encargado de multiplicar las neuronas de entrada por su peso correspondiente, debido a que existen dos posibles entradas. Este multiplexor cuenta con entradas y salidas registradas mediante un reloj, al ser 784 neuronas de entrada y 10 en la capa oculta, esto supondría un aumento de 7840 ciclos. Lo que supondría un total de 21168 ciclos, 104272 ciclos menos respecto a la anterior versión se ahorrarían 1.04ms.

Para llevar a cabo esta solución, la red seguirá la misma ejecución que la versión anterior, pero con algunos cambios. Cuando la señal de start sea igual a 1, el controlador de la

memoria de los pesos activará la salida valid, para indiciar que se puede comenzar el preprocesado de datos. Esta salida, estará conectada a la entrada valid_in del bloque mux_input.

Durante el cálculo de la primera neurona de la capa oculta, es necesario hacer el preprocesado de datos, pero después ya no es necesario porque los valores se encuentran guardados en la memoria de la capa de entrada. El bloque mux_input, es el encargado de seleccionar cuándo se debe realizar el preprocesado de los datos de entrada y cuándo no es necesario. La entrada enable_input estará conectada a la salida enable_mux_input del controlador de la memoria de la capa de entrada. Esta salida, se activará únicamente cuando la dirección de la memoria de la capa de entrada haya alcanzado su máximo valor. Una vez que se active, permanecerá activa durante el resto de la ejecución. De esta forma, conoceremos cuando se ha recorrido por primera vez la memoria de entrada. Cuando esto ocurra, se habrán guardado el preprocesado de los datos, por lo que no será necesario volverlos a calcular. Cuando la entrada valid_in del bloque mux_input sea igual a 1, significa que es necesario realizar una operación y dependiendo de si enable_input es igual a 1 o no, sabremos si será necesario realizar el preprocesado de datos. En el caso de que no sea igual a 1, se activará la salida gain. Esta salida está conectada a las entradas tvalid del floating_point_6, encargado de realizar el preprocesado de los datos de entrada.

Una vez realizado el cálculo de preprocesado de datos en el bloque floating_point_6, se guardará el valor en la memoria de las neuronas de la capa de entrada sobrescribiendo su valor, por este nuevo resultado. El resultado de esta operación será conectado a la entrada d de la memoria de la capa de entrada y la salida result_tvalid será conectada a la entrada w de esta memoria. Cuando se haya realizado la operación, la salida result_tvalid será igual a 1, lo que implica que la entrada we de la memoria será igual a 1. Esta entrada es la habilitación de escritura de la memoria; por lo que cuando sea 1, se sobrescribirá la dirección de memoria seleccionado por el valor presente en la entrada d. A su vez, la salida result_tvalid, será conectada a la entrada valid_gain del bloque mux_input.

La entrada valid_gain del módulo mux_input, será igual a 1, cuando se haya realizado el preprocesado de datos de forma correcta. Cuando esto ocurra, será necesario multiplicar este valor por el peso correspondiente a la neurona de entrada. Si esto ocurre, la señal valid_gain será igual a 1 y cuando haya un flanco de subida del reloj, la salida valid de este bloque será igual a 1 durante un ciclo de reloj. Esta salida estará conectada a las entradas tvalid del bloque floating_point_0 encargado de realizar la operación de multiplicar los datos de entrada por el peso y sumarle el resultado anterior.

Una vez que esto ocurra la red seguirá su ejecución de forma normal, como se ha explicado anteriormente.

Terminado el preprocesado de datos para todas neuronas de entradas, se reiniciará el contador del controlador de la memoria de la capa de entrada, activando la salida enable_mux_input del controlador. Esta salida irá conectada a la entrada enable_input del bloque mux_input.

Cuando la entrada enable_input del bloque mux_input sea igual a 1, la salida valid de este bloque será igual a 1, durante un ciclo de reloj. Esta salida está conectada a la entrada tvalid del floating_point_0, encargado de multiplicar las neuronas de entrada por su peso correspondiente. Por lo que se realizará directamente esta operación sin hacer el preprocesado de los datos, ahorrándonos ciclos de reloj. El resto del programa seguirá la secuencia explicada anteriormente. Esta será la versión definitiva de la red implementada.

3.4.2. Tablas de aproximación (Look up Tables)

Esta mejora consiste en crear un proyecto aparte, a partir de esta última versión implementada. Donde el cálculo de las funciones de activación se realice a través de tablas de aproximación. Esto consiste en guardar los valores de la función, de los puntos que se consideren más significativos. En esta red, las tablas de aproximación solo se pueden llevar a cabo con la función tangente hiperbólica. La función softmax, depende de la suma de todas las exponenciales y que se puede acumular el error. Esta función no consume tanto tiempo ni recursos como la función tangente hiperbólica.

Para realizar la función tangente hiperbólica son necesarios 5 bloques floating point encargados de realizar las distintas operaciones. Siendo necesarios un total de 78 ciclos de reloj, para llevar a cabo esta operación debido a las latencias de los distintos bloques. Sin embargo, para realizar la función exponencial de la función softmax, solo es necesario un bloque, lo que supone 20 ciclos de reloj. El resto de las operaciones como la suma de exponenciales y la división de la función softmax, habría que realizarlas igualmente. Lo único que se podría realizar con la ayuda de tablas de aproximación de la función softmax, es la exponencial, lo que no supone un ahorro significativo.

Si realizamos el cálculo de la función tangente hiperbólica mediante tablas de aproximación, únicamente sería necesario un bloque floating point, el cual tardaría 6 ciclos de reloj en realizar la operación. Esto supone un ahorro significativo tanto en tiempo de ejecución como recursos. Para llevar a cabo esta mejora se ha realizado los siguientes cambios respecto al proyecto original:

Los cambios implementados en esta mejora solo afectan a los bloques encargados de realizar el cálculo de la función tangente hiperbólica, la cual sigue la siguiente ecuación $tanh = \frac{2}{1+e^{-2\cdot x}} - 1$.

El resto de los procesos de la red se ejecutarán de la forma explicada anteriormente. Los cambios introducidos empiezan a partir de que la salida enable_activation_hidden del bloque multiplexor_memory_hidden sea igual a 1. Esta salida, es la encargada de activar la operación de la función de activación, en este proyecto esta salida estará conectada a la entrada tvalid del bloque floating_point_8.

El bloque floating_point_8, será el encargado de transformar los datos con los que opera la red de coma flotante a coma fija. La finalidad es que sea más sencillo a la hora de saber cuál es el valor de entrada x de esta función y poder realizar una mejor aproximación. A la entrada A de este bloque estará conectada la salida de la memoria de la capa oculta, de forma que cuando la entrada tvalid sea igual a 1, se realizará una conversión a coma fija del valor encontrado en la memoria de la capa oculta. El resultado de esta conversión será un vector de 16 bits, el cual solo tendrá parte entera, donde el primer bit del vector corresponde al signo, 0 si es negativo 1 si es positivo. Es decir, el resultado será redondeado; por ejemplo, si la conversión a coma fija da un resultado de 7,6, el resultado obtenido a la salida de este bloque será igual a 7 representado en un vector de 16 bits. Esta salida será conectada a la entrada din del módulo lookup diseñado. La salida result_tvalid encargada de confirmar que se ha realizado la operación, será conectada a la entrada valid del bloque lookup.

El módulo lookup será el encargado de almacenar los valores de la tabla de aproximación y seleccionar el valor más adecuado según la entrada proporcionada. A continuación, en la tabla 2, se muestra ejemplo de una tabla de aproximación con los valores más significativos:

x	tanh(x)	tanh(x) (floating point)
<-4	-1	0xbf800000
-4	-0,9993293	0xbf7fd40c
-3	-0,9950548	0xbf7ebbe9
-2	-0,9640276	0xbf76ca83
-1	-0,7615942	0xbf42f7d6
0	0	0x00000000
1	0,7615942	0x3f42f7d6
2	0,9640276	0x3f76ca83
3	0.9950548	0x3f7ebbe9
4	0,9993293	0x3f7fd40c
5	0,9999092	0x3f7ffa0d
>5	1	0x3f800000

Tabla 2: Tabla de aproximación (Look up table).

Esta tabla puede ser modificada por el usuario añadiendo más o menos valores, lo que le daría más precisión al cálculo de la función, al igual que si se realizara la conversión con decimales. Con esta tabla se ha conseguido un resultado correcto de la red.

El funcionamiento del bloque es el siguiente: Cuando la entrada valid sea igual a 1, se comprobará si el dato de entrada din es mayor que 6 y menor o igual que 32767, el cual es el máximo número positivo que puede alcanzar este resultado. A partir de que el vector tenga un valor de 32768 (1000000000000000 o “0x8000”) tendrá un 1 en el primer bit,

por lo que se consideraría como un número negativo. Si la condición mencionada anteriormente se cumple, la salida dout el cual será un vector de 32 bits, que tomará el siguiente valor “0x3f800000”, igual a 1 en representación decimal. Este valor debe estar representado en coma flotante de nuevo, debido a que será el resultado de la función, será guardado en la memoria de la capa oculta y utilizado posteriormente por la red.

Si la condición mencionada anteriormente no se cumple, significaría que el número es menor que 6. En ese caso, se realizará un case con el dato din. Dependiendo del valor de din, la salida dout tomará el valor correspondiente a los datos encontrados en la tabla 2. Este dato será conectado a la entrada din_activation del bloque multiplexor_memory hidden, encargado de multiplexar la memoria de la capa oculta.

A su vez, si valid es igual a 1, la salida ok será igual a 1. Esta salida servirá como confirmación de la operación y será conectada a las entradas we_activation y valid_activation del bloque multiplexor_memory_hidden.

Como ya sabemos, el bloque multiplexor_memory_hidden se encargará de guardar el dato apropiado en la memoria de la capa oculta y mandar las respectivas señales para que continúe la ejecución de la red de la misma manera a como se había explicado anteriormente.

Como podemos observar, para seleccionar los valores guardados no es necesario ningún ciclo de reloj. Únicamente será necesario emplear ciclos de reloj en la transformación de coma flotante a coma fija, donde se invertirán 6 ciclos como se ha mencionado. Dado que es necesario realizar esta operación 10 veces, supondría 60 ciclos de ejecución durante la ejecución de toda la red. Invirtiendo un tiempo total de 0.6µs en realizar el cálculo mediante tablas de aproximación.

En el caso de no utilizar la tabla de aproximación, harían falta 78 ciclos de reloj para calcular la función de activación. Dado que la operación se debe realizar 10 veces, supondría un total de 780 ciclos de reloj. Con un tiempo total de 7.8µs. Es decir, se ahorran 720 ciclos de reloj lo que supone 7.2µs. En el apartado de análisis y resultados, se procederá a analizar los resultados ofrecidos por esta red, analizando su precisión.

3.5. Resultados

Se mostrarán los distintos resultados obtenidos por los dos tipos de redes. La red que realiza el cálculo de la función de activación, y la red que hace uso de tablas de aproximación para hallar el valor de esta función. Introducimos como entrada de la red, distintas imágenes en las cuales aparecen representados todos los dígitos del 0 al 9. También, se mostrarán los resultados obtenidos por Matlab para esas mismas entradas, así como el tiempo de ejecución de cada red.

3.5.1. Resultados obtenidos en Matlab

En la tabla 3, podemos encontrar los distintos resultados proporcionados por la red ejecutada en Matlab, para distintas entradas. En las columnas encontramos el valor de cada neurona de salida, encargadas de representar cada posible solución a la red. En cada fila se muestra el número que aparece en la imagen. Se han seleccionado 10 imágenes en las cuales, se encuentran todos los dígitos posibles.

Número	0	1	2	3	4	5	6	7	8	9
0	0,994960148	8,18E-07	0,000194415	0,00044245	7,20E-06	0,003291365	0,000362935	2,53E-05	0,000385151	0,000330227
1	1,40E-05	0,997049764	0,002104542	0,000347967	5,77E-06	6,53E-05	0,000124034	0,000170184	0,000109768	8,62E-06
2	0,000286166	0,000288813	0,986908933	0,000383334	1,95E-05	0,004640936	0,000228755	3,99E-06	0,007236378	3,24E-06
3	0,000228724	0,000199261	0,009977253	0,910587514	4,33E-05	0,066807195	0,001056793	4,97E-06	0,011076752	1,82E-05
4	0,000221685	1,18E-06	8,29E-05	3,84E-06	0,997291107	0,000536729	0,001085928	0,000214263	0,000185541	0,000376842
5	0,000256273	3,58E-06	1,34E-05	0,037599978	8,54E-05	0,935850865	3,50E-05	2,61E-05	0,026085333	4,42E-05
6	0,000628109	0,000177735	0,00467414	0,01787901	0,004864386	0,000328289	0,970841612	9,98E-05	0,000368391	0,000138555
7	2,18E-05	1,82E-06	4,85E-05	7,07E-04	5,21E-05	4,34E-03	1,87E-07	9,95E-01	2,82E-05	0,000289038
8	3,31E-05	6,60E-06	0,002060864	0,000381158	0,000111177	0,002601722	5,70E-06	5,56E-06	0,994631433	0,000162659
9	4,12E-05	0,000951205	8,26E-05	0,000702147	0,00711724	0,000417256	0,000107549	0,003126404	0,002762822	0,984691555

Tabla 3: Resultados implementación en Matlab

En cuanto al tiempo de ejecución, al ser un programa software, el cual será ejecutado en un ordenador, el tiempo de ejecución de la red depende de diversos factores, como la capacidad de procesamiento del ordenador, la cantidad de tareas que se estén ejecutando a la vez. Para hallar el tiempo, se ha utilizado la función tictoc en Matlab, la cual te permite saber el tiempo empleado en ejecutar el código. Se han conseguido diversos tiempos ya que el tiempo de ejecución no es constante y varía en cada dígito. Los tiempos tomados a la hora de ejecutar la red con las 10 entradas distintas, están comprendidos entre los de 2,5ms y un máximo de 10ms. Datos obtenidos en un ordenador con un procesador Intel Core i7- 4720HQ, con una frecuencia de 3,6GHz.

3.5.2. Resultados de la red neuronal implementada en la FGPA.

A continuación, se mostrarán los resultados obtenidos para la red neuronal diseñada, realizando el cálculo de la función de activación; es decir, sin usar tablas de aproximación. Los datos se han obtenido mediante la simulación del programa vivado. Éste nos proporciona los datos almacenados en cada memoria, así como el tiempo de ejecución de la red. Gracias a esto, podemos ser capaces de tomar datos, debido a que una vez implementada en la FPGA, únicamente somos podemos observar cómo se encienden los leds cuando pulsamos el botón de start, sin llegar a conocer el valor de cada neurona de salida y el tiempo de ejecución.

Se han seleccionado exactamente los mismos datos de entrada que en el programa Matlab, los resultados obtenidos en cada neurona de salida son mostrados en la tabla 4.

Número	0	1	2	3	4	5	6	7	8	9
0	0,99528134	8,10E-07	1,96E-04	4,45E-04	7,17E-06	0,003295894	3,60E-04	2,54E-05	3,87E-04	3,33E-04
1	1,33E-05	0,99729174	0,001919774	3,18E-04	5,61E-06	6,14E-05	1,23E-04	1,59E-04	1,01E-04	7,71E-06
2	2,86E-04	2,89E-04	0,98690915	3,83E-04	1,94E-05	0,00464081	2,29E-04	3,99E-06	0,007236331	3,24E-06
3	0,09321856	1,97E-04	0,009981253	0,9108259	4,30E-05	0,066583805	0,001045508	4,95E-06	0,011072542	1,83E-05
4	1,99E-04	1,32E-06	6,34E-05	2,95E-06	0,9974817	4,65E-04	0,001197512	1,79E-04	1,46E-04	2,65E-04
5	2,56E-04	3,58E-06	1,34E-05	0,03760479	8,54E-05	0,93584347	3,49E-05	2,61E-05	0,026087988	4,42E-05
6	6,27E-04	1,78E-04	0,004652301	0,017796766	0,004858748	3,27E-04	0,9709566	9,94E-05	3,67E-04	1,38E-04
7	2,18E-05	1,82E-06	4,85E-05	7,07E-04	5,21E-05	4,34E-03	1,87E-07	9,95E-01	2,82E-05	2,89E-04
8	3,31E-05	6,60E-06	0,002060857	3,81E-04	1,11E-04	0,002601676	5,70E-06	5,56E-06	0,99463147	1,63E-04
9	2,29E-03	4,13E-05	4,74E-05	3,90E-04	6,80E-04	1,05E-04	5,35E-06	0,00133298	9,91E-04	0,9963984

Tabla 4: Resultados implementación FPGA.

En la tabla 5, se muestran los recursos utilizados para la implementación de la red neuronal en la placa Zybo (Zynq-7000). Podemos observar que el recurso más utilizado es la memoria RAM, siendo usado el 92.63% de la memoria disponible. Encontramos los LUTs como el segundo recurso más usado, donde son usados el 66.83% de los LUTs disponibles.

Resource	Utilization	Available	Utilization %
LUT	11762	17600	66.83
LUTRAM	5561	6000	92.68
FF	8600	35200	24.43
DSP	18	80	22.50
IO	7	100	7.00
BUFG	1	32	3.13

Tabla 5: Recursos utilizados implementación estándar.

En cuanto el tiempo de ejecución, presenta un tiempo constante debido a que viene dado por los ciclos de reloj de la placa. Con una frecuencia de 100MHz, el tiempo total de ejecución de la red es de 1,578205ms.

El proyecto cumple las especificaciones de tiempo impuestas, el worst negative slack es igual a 0,717ns. Lo que quiere decir, que del periodo de 10ns que tiene el reloj, el camino crítico o el camino de máximo retardo entre dos registros, es de 9,283ns, y por lo tanto, le sobran 0,717ns. Si cambiáramos la frecuencia del reloj a 9,283ns, la cual sería la frecuencia máxima a la que se podría ejecutar esta red. El tiempo de ejecución de la red pasaría a ser 1,4565ms.

3.5.3. Resultados de la red neuronal mediante tablas de aproximación.

A continuación, se mostrarán los resultados obtenidos en la capa de salida de la red neuronal, la cual realiza el cálculo de la función de activación mediante tablas de aproximación. Los resultados serán obtenidos de la misma forma que en el apartado anterior, mediante la simulación del programa vivado. El programa nos permite visualizar el resultado almacenado en la memoria de la capa de salida, el resultado mostrado estará representado en el estándar IEEE 754, por lo que será transformado a coma fija y representación decimal para una correcta comprensión.

En la tabla 6 se mostrarán los resultados obtenidos en las neuronas de salidas correspondientes:

Número	0	1	2	3	4	5	6	7	8	9
0	0,99512863	7,12E-07	1,62E-04	4,62E-04	6,21E-06	0,003451996	4,16E-04	2,41E-05	3,46E-04	2,94E-04
1	1,56E-05	0,9949316	0,002302035	6,03E-04	2,86E-05	2,84E-04	1,96E-04	5,70E-04	1,05E-03	2,01E-05
2	2,78E-04	2,00E-04	0,9879223	2,49E-04	1,75E-05	3,35E-03	2,63E-04	2,69E-06	0,007713208	3,19E-06
3	1,54E-04	1,55E-04	0,007926497	0,89096975	6,41E-05	0,08474293	0,001067848	1,06E-05	0,014881397	2,79E-05
4	3,55E-04	3,68E-06	2,41E-04	3,87E-06	0,9961021	1,97E-04	0,002487428	1,44E-04	8,51E-05	3,81E-04
5	2,19E-04	7,89E-06	3,91E-05	0,06581307	6,84E-05	0,8876055	4,56E-05	3,78E-05	0,04612395	3,96E-05
6	6,71E-04	1,39E-04	0,003853709	0,011807301	0,005741295	2,89E-04	0,9770367	7,45E-05	2,42E-04	1,46E-04
7	3,29E-05	5,89E-06	1,56E-04	2,93E-03	5,64E-05	7,42E-03	3,56E-07	9,89E-01	9,43E-05	9,43E-05
8	1,65E-05	7,10E-06	0,001293711	2,08E-04	6,95E-05	8,89E-04	3,07E-06	5,20E-06	0,99728423	2,24E-04
9	0,002838187	6,53E-05	8,16E-05	2,41E-04	5,03E-04	1,86E-04	1,29E-05	0,001094493	0,001999937	0,9958037

Tabla 6: Resultados implementación mediante tablas de aproximación.

En la tabla 7, se encuentran los recursos utilizados de la FPGA a la hora de implementar esta red. Podemos observar que el recurso más utilizado es la memoria RAM usando un 91.07% de la memoria disponible. En segundo lugar, podemos encontrar LUTs, donde se hace uso del 56,44% de los LUTs disponibles.

Resource	Utilization	Available	Utilization %
LUT	9933	17600	56.44
LUTRAM	5464	6000	91.07
FF	6057	35200	17.21
DSP	11	80	13.75
IO	7	100	7.00
BUFG	1	32	3.13

Tabla 7: Recursos utilizados implementación con tablas de aproximación.

El tiempo de ejecución de esta red también es constante, debido a estar implementado en una FPGA. Con una frecuencia de 100MHz, el tiempo total de ejecución, es igual a 1,571005ms.

3.6. Análisis

Podemos observar en los resultados mostrados anteriormente, que la red diseñada cumple el objetivo del proyecto. Es capaz de reconocer los dígitos representados en una imagen mediante el valor de los distintos píxeles que componen dicha imagen. Se llega a obtener una probabilidad mayor del 90% en todos los casos. Con un porcentaje máximo de 99% en dígitos como el 0, 1, 4 y 8. En el peor de los casos, encontramos el número 3 con una probabilidad del 91.08%, como podemos observar en la tabla 4.

Si comparamos la red ejecutada en Matlab con la implementada en la FPGA, podemos observar que los resultados obtenidos son prácticamente idénticos.

A la hora de comparar los resultados de cada neurona, se observan ligeras variaciones en algunas ocasiones, llegando a tener un error de 0,09298984, en los casos más extremos. Estas variaciones al ser tan pequeñas no afectan ni al funcionamiento de la red ni al resultado de las neuronas. Este error, puede ser ocasionado por errores de aproximación a la hora de convertir los números de coma flotante a como fija. También puede estar ocasionado por la transformación de los datos de 64 a 32 bits. Todos los datos proporcionados por Matlab se encuentran en presión double, 64 bits. Al ser magnitudes de error tan pequeñas, en algunos casos, podemos observar que el resultado proporcionado por la red es idéntico que la solución proporcionada por Matlab.

En la tabla 8, se representa el error obtenido respecto a Matlab en valor absoluto, Como podemos ver, el error obtenido se debe a una pérdida de precisión, llegando a un máximo de 0,09298984 y un mínimo de $3{,}74765 \cdot 10^{-13}$. El error medio es igual a 0,001207759.

Número	0	1	2	3	4	5	6	7	8	9
0	0,00032119	7,5233E-09	1,2389E-06	2,7366E-06	3,21942E-08	4,5288E-06	3,0454E-06	7,0066E-08	2,0505E-06	3,2477E-06
1	7,4161E-07	0,00024198	0,000184768	3,0194E-05	1,58739E-07	3,9551E-06	5,6494E-07	1,1618E-05	9,0169E-06	9,1505E-07
2	9,4613E-09	2,5365E-08	2,17037E-07	6,5969E-10	1,20925E-09	1,257E-07	1,9726E-08	7,3989E-11	4,7467E-08	6,5082E-11
3	0,09298984	2,2488E-06	4,0002E-06	0,00023839	3,34861E-07	0,00022339	1,1285E-05	1,141E-08	4,2103E-06	5,4589E-08
4	2,3059E-05	1,4705E-07	1,94458E-05	8,8752E-07	0,000190593	7,2141E-05	0,00011158	3,508E-05	3,9817E-05	0,00011186
5	5,9655E-10	1,0362E-09	1,77114E-09	4,8124E-06	1,4113E-08	7,3949E-06	9,5842E-09	8,1102E-10	2,6549E-06	9,8212E-09
6	1,553E-06	6,8238E-08	2,18393E-05	8,2244E-05	5,63791E-06	1,0003E-06	0,00011499	3,4907E-07	1,5905E-06	7,9933E-07
7	6,2757E-09	2,7125E-12	1,73038E-12	5,6473E-10	1,01607E-10	3,4175E-09	3,7476E-13	2,6519E-08	7,7256E-13	4,6374E-10
8	5,0399E-09	4,2583E-11	7,78151E-09	2,9171E-09	1,2858E-09	4,5432E-08	8,6363E-11	1,1328E-11	3,7031E-08	1,0109E-09
9	0,00224915	0,00090987	3,52275E-05	0,00031192	0,00643692	0,00031227	0,0001022	0,00179342	0,00177204	0,01170685

Tabla 8: Error respecto a Matlab.

Respecto al tiempo de ejecución, se observa que en ciertas ocasiones Matlab es capaz de ejecutar la red en un tiempo mínimo de 2,5ms y un máximo de 10ms.

Presentando grandes variaciones de una ejecución a otra, si realizamos la media con el tiempo de ejecución empleado por cada entrada, se obtiene un valor medio de 3,6895ms. Sin embargo, la implementación realizada en la FPGA se caracteriza por tener un tiempo constante, el cual es igual a 1,578255ms. Podemos ver que la red neuronal implementada en la FPGA es más rápida que la red implementada en Matlab.

A la hora de comparar las dos versiones implementadas en la FPGA, podemos observar que la red que hace uso de las tablas de aproximación presenta distintos valores en todas las neuronas de la capa de salida, teniendo error medio de 0,00278941.

Como vemos, está perdida de precisión no es significativa a la hora de reconocer la imagen representada en la red, llegando a distinguir entre los distintos dígitos con una probabilidad mayor del 88% en todos los casos. Con un 99% en dígitos como el 0, 1, 8 y 9 y un 88,76% en el caso del número 5.

Esta pérdida de precisión como ya sabemos se debe a que se realiza una aproximación de la función tangente hiperbólica. En la figura 21, podemos observar la función tangente hiperbólica dibujada en azul mediante una línea discontinua y en rojo la función obtenida al unir los distintos puntos de la tabla de aproximación. Se aprecia que en los intervalos de x [-1,0] y [0,1], es donde se pierde más precisión. Esto se podría solucionar añadiendo puntos intermedios entre esos intervalos en la tabla de aproximación.

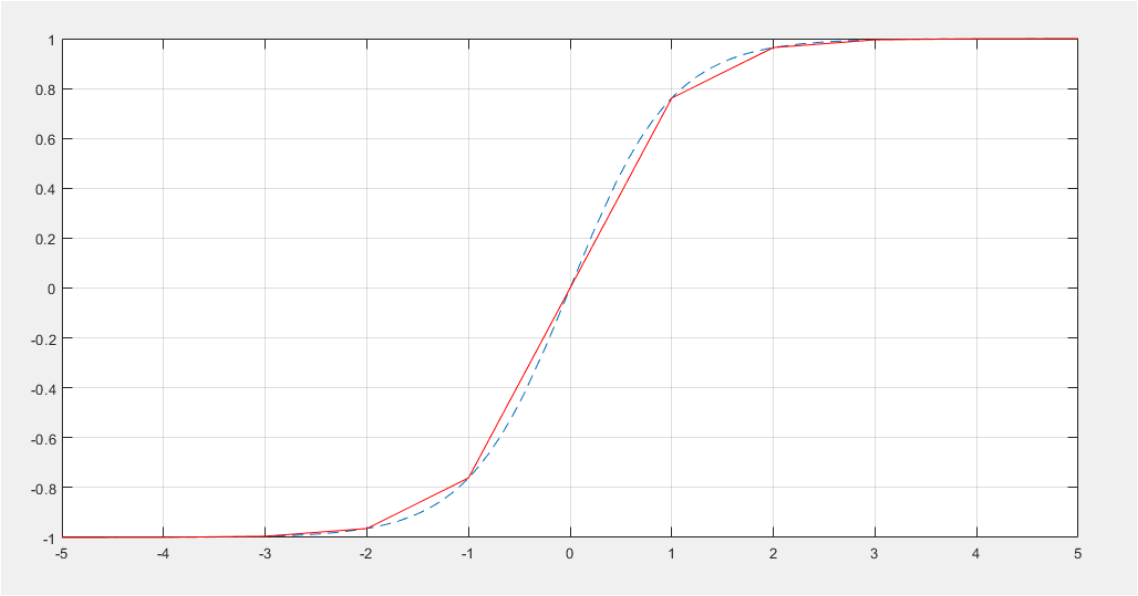


Figura 21: Comparativa entre tablas de aproximación y tangente hiperbólica.

Las tablas de aproximación se pueden rellenar al gusto del usuario, por lo que cuantos más puntos significativos de la función contenga la tabla, se obtendrá más precisión. Reduciendo el error a la hora de ejecutar la red neuronal, pero también se debe tener en cuenta, que cuantos más puntos de la función se registren en la tabla, será necesario un mayor espacio en la memoria para almacenar estos valores.

En la tabla 9 podemos observar, el error en valor absoluto, presente en las neuronas de salida a la hora de realizar el cálculo de la función tangente hiperbólica mediante tablas de aproximación. Llegando a un error máximo de 0,09306454, un mínimo de $5{,}23464 \cdot 10^{-08}$ y el error medio es igual a 0,00278941.

Número	0	1	2	3	4	5	6	7	8	9
0	0,00015271	9,8083E-08	3,32371E-05	1,7108E-05	9,55992E-07	0,0001561	5,6597E-05	1,3069E-06	4,1532E-05	3,9313E-05
1	2,3166E-06	0,00236014	0,000382261	0,00028496	2,3003E-05	0,0002222	7,2544E-05	0,00041109	0,00094943	1,2365E-05
2	8,3262E-06	8,8893E-05	0,00101315	0,00013448	1,99858E-06	0,0012889	3,4025E-05	1,2943E-06	0,00047688	5,2346E-08
3	0,09306454	4,2075E-05	0,002054756	0,01985615	2,10953E-05	0,01815913	2,234E-05	5,6726E-06	0,00380886	9,605E-06
4	0,000156	2,3592E-06	0,000177151	9,1686E-07	0,0013796	0,00026713	0,00128992	3,5212E-05	6,0659E-05	0,00011637
5	3,7247E-05	4,3165E-06	2,57518E-05	0,02820828	1,69455E-05	0,04823797	1,066E-05	1,1705E-05	0,02003596	4,5663E-06
6	4,4267E-05	3,9186E-05	0,000798592	0,00598947	0,000882547	3,7949E-05	0,0060801	2,4962E-05	0,00012465	7,8837E-06
7	1,1148E-05	4,0704E-06	0,000107896	0,00222114	4,3193E-06	0,00307547	1,6818E-07	0,00554436	6,6148E-05	0,00019473
8	1,6658E-05	5,0038E-07	0,000767146	0,00017302	4,16577E-05	0,00171268	2,6302E-06	3,5898E-07	0,00265276	6,0942E-05
9	0,0005478	2,3918E-05	3,42716E-05	0,00014903	0,000177656	8,1312E-05	7,5122E-06	0,00023849	0,00100916	0,0005947

Tabla 9: Error obtenido al usar tablas de aproximación

En cuanto al tiempo de ejecución podemos observar que se disminuye el tiempo de ejecución en $7.2 \cdot 10^{-3}$ ms. No es una reducción muy significativa debido a que solo se realiza el cálculo de una única función mediante tablas de aproximación, la cual solamente se ejecuta un total de 10 veces.

Por otro lado, si comparamos los recursos utilizados por la red neuronal estándar con los recursos utilizados por la red con tablas de aproximación, nos encontramos que la red que hace uso de las tablas de aproximación consume muchos menos recursos, reduciendo un 8,75% los DSP.

Los DSP son utilizados principalmente por los bloques floating point a la hora de realizar las operaciones, por lo que al hacer un uso menor de bloques floating_point en esta implementación se reducen los DSP utilizados. Se reduce un 7,22% el uso de biestables, y el uso de los LUTs un 10,39%. También podemos observar una reducción del 1,76% de la memoria RAM utilizada.

Por último, a la hora de comparar la potencia podemos encontrar que, a la hora de realizar la implementación estándar de la red, la FPGA consume una potencia total de 0.385W. En cambio, a la hora de realizar la implementación de la red neuronal mediante tablas de aproximación, la FPGA consume una potencia de 0,325W. La implementación mediante tablas de aproximación reduce un 0,06 W la potencia consumida por la FPGA.

3.7. Planificación y presupuesto

En este apartado, se explicará la planificación escogida para realizar las distintas tareas que componen el proyecto, así como su presupuesto.

3.7.1. Planificación

Para llevar a cabo este proyecto han sido necesarias varias tareas, las cuales son:

- **Investigación y documentación:** Antes de comenzar con el diseño de la red, es necesario recopilar toda la información precisa sobre las redes neuronales, estudiar las distintas implementaciones y las aplicaciones de los distintos dispositivos.
- **Desarrollo de ideas:** Una vez se ha investigado sobre las redes neuronales, habrá que definir el alcance del proyecto y seleccionar la tarea que tendrá que desempeñar la red.
- **Implementación software y entrenamiento:** Definido el alcance del proyecto, es necesario buscar una herramienta para entrenar la red, dado que el entrenamiento no va a ser desarrollado en lenguaje hardware.

- **Implementación Hardware:** Cuando se hayan realizado las tareas anteriores, se puede comenzar con la implementación hardware de la red entrenada previamente.
- **Mejoras:** Conseguida una primera versión de la red que cumpla el objetivo propuesto, se podrán llevar a cabo las mejoras pertinentes, para dar un valor adicional a la red implementada.
- **Pruebas y toma de resultados:** Consiste en ejecutar la red con distintas entradas, comprobando la fiabilidad de la red. Así como recoger los resultados proporcionado por cada neurona y transformarlos de coma flotante a coma fija.
- **Memoria:** Por último, una vez terminado el proyecto hay que documentar todas las tareas llevadas a cabo y resultados obtenidos en una memoria.

En la figura 22, aparece representado un diagrama de Gantt de la planificación del trabajo. Se muestra el orden en el que se ha realizado cada tarea y el tiempo empleado para llevar a cabo cada una de ella, donde el eje horizontal representa las horas invertidas en el proyecto y el eje vertical cada una de las tareas llevadas a cabo.

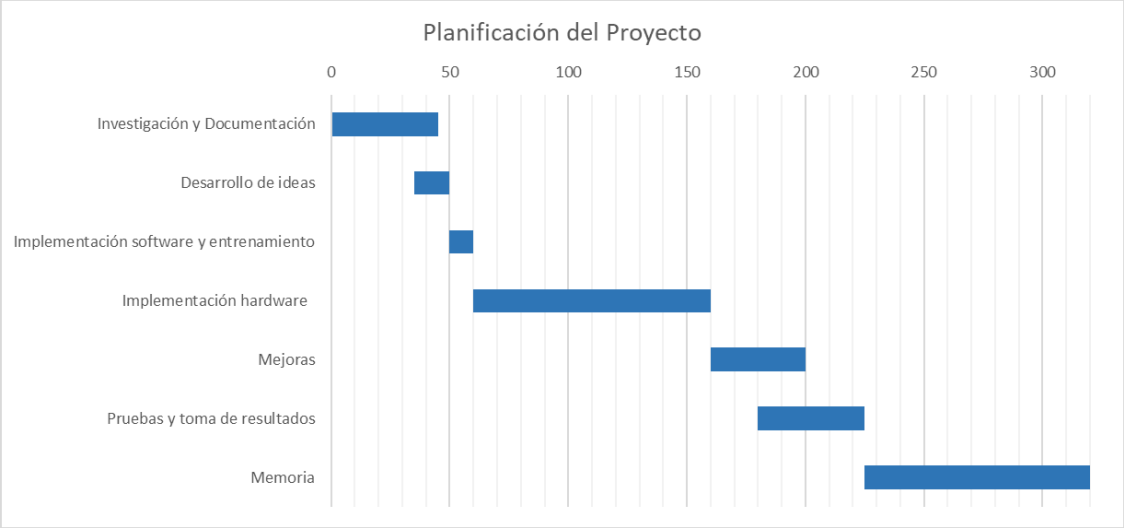


Figura 22:Planificación del proyecto, diagrama de Grantt.

El total de horas invertidas para realizar este proyecto son 320 horas. Como podemos ver, hay tareas que se desarrollan al mismo tiempo como las pruebas y toma de resultados o el desarrollo de ideas.

3.7.2. Presupuesto

Los materiales necesarios para la elaboración de este proyecto son los siguientes:

- **FPGA:** Para llevar a cabo este proyecto, se necesita una FPGA donde poder implementar la red neuronal. En este proyecto, se ha hecho uso de una FPGA de la marca Zybo, modelo Zynq-7000.
- **Ordenador:** Es necesario un ordenador capaz de ejecutar los programas requeridos para la realización de este trabajo. Con el fin de desarrollar la implementación software de la red, y posteriormente el desarrollo de la implementación hardware.
- **Licencia Matlab:** Para hacer uso del programa Matlab, es obligatorio tener una licencia, la cual hay que renovar anualmente. En este caso, la licencia necesaria para la realización de este trabajo ha sido suministrada por la Universidad.
  En caso contrario, sería necesario comprar una licencia. El precio de estas licencias varía según la finalidad con la que se vaya a utilizar el programa y quien sea la persona que la esté utilizando. Para la elaboración del presupuesto hemos escogido una licencia de estudiante.
- **Licencia Xilinx:** La licencia Xilinx nos permite hacer uso del programa vivado, para la programación de la FPGA.
  Para poder usar esta herramienta es necesario comprar una licencia, al igual que Matlab, la licencia utilizada en este proyecto ha sido proporcionada por la Universidad. Para la elaboración de este presupuesto, se ha incluido el precio de una licencia estándar.

La tabla 10, muestra las herramientas utilizadas durante este proyecto y el coste de cada una. Con un importe total de 3.4538 euros.

Producto	Cantidad (Ud)	Precio unitario
Zybo Zinq-7000, FPGA	1	167 €
Portátil Asus R560UD	1	549 €
Matlab Student Suite license	1	69 €
Vivado HL Design Edition	1	2.653 €
TOTAL		**3.438 €**

Tabla 10: Presupuesto del proyecto.

4. Conclusión

En este trabajo se ha implementado una red neuronal funcional en una FPGA. La red creada es capaz de reconocer los dígitos que se encuentran en una imagen, mediante los datos proporcionados por los pixeles que componen la imagen.

La red diseñada presenta una pequeña pérdida de precisión respecto a Matlab, pero nada significativo. Es capaz de desempeñar con éxito, la tarea para la que ha sido creada, obteniendo más de un 90% de probabilidad al reconocer el dígito correcto. Gracias a la implementación de la red neuronal en una FPGA, hemos conseguido un tiempo de ejecución constante de 1,5ms. Siendo capaz de realizar la ejecución a una mayor rapidez que el programa Matlab, ofreciendo una alternativa fiable, rápida y económicamente asequible frente la implementación en otros dispositivos.

Se han conseguido llevar a cabo las diversas mejoras propuestas creando una red neuronal, la cual es capaz de hallar el valor de la función de activación mediante tablas de aproximación. Aportando un pequeño ahorro en el tiempo de ejecución, $7{,}2 \cdot 10^{-3}$ms. A su vez, esta mejora nos permite un ahorro de los recursos de la FPGA destinados a la ejecución de la red, así como un ahorro en la potencia consumida por este dispositivo.

Se reduce la precisión en cada neurona de salida, pero la red sigue siendo capaz de distinguir correctamente el dígito mostrado en la imagen y se otorga a la neurona de salida una probabilidad del 88% respecto al resto de neuronas encontradas en la capa de salida. Dado que el propósito de esta red es distinguir el número representado en la imagen, está solución nos permite seguir cumpliendo con el objetivo.

Este trabajo refleja que los dispositivos FPGA, pueden ser candidatos a tener en cuenta a la hora de elaborar redes neuronales sin mucha complejidad.

Un ejemplo serían las redes enfocadas al reconocimiento de imágenes con baja resolución, 28x28 píxeles, siendo un factor limitante la memoria de estos dispositivos, debido a se almacena una gran cantidad de datos. La capacidad de procesamiento también es un factor limitante, ya que son necesarias realizar una gran cantidad de operaciones en coma flotante.

Gracias a este trabajo se han obtenido amplios conocimientos sobre el funcionamiento de las redes neuronales, y el diseño de circuitos integrados.

4.1. Líneas futuras

Por falta de tiempo, no se han podido llevar a cabo todas las ampliaciones pensadas para este proyecto. Por ello, a continuación, se enumeran las posibles mejoras que se podrían llevar a cabo en un futuro, estableciendo las líneas futuras de este trabajo:

- **Mejoraras en la entrada de datos**: En el proyecto realizado, los valores de cada píxel encargados de formar la imagen eran almacenados previamente en una memoria antes de comenzar la ejecución. Es decir, antes era necesario establecer los valores por defecto de la memoria, después sintetizar el proyecto y programarlo en la FPGA, lo que nos podría llevar varios minutos.
  Si se consiguiera transmitir los valores de entrada de la red mediante una conexión, como, por ejemplo, una conexión UART. La FPGA sería capaz de comunicarse con el ordenador y adquirir directamente los datos del ordenador y guardados en memoria. Así se permite la entrada de datos en la red de una manera más flexible.

- **Realizar el entrenamiento de la red:** Realizar el entrenamiento de la red en hardware, sin la necesidad de realizar una implementación previa en software para obtener los datos del entrenamiento. Esto se podría realizar mediante una máquina de estados, donde un estado sería el encargado de realizar el entrenamiento y otro estado sería el encargado de la ejecución de la red.

- **Permitir una mayor resolución de las imágenes:** La red neuronal creada está pensada para imágenes con unas dimensiones de 28x28 píxeles. Si se consiguiese aumentar las dimensiones de las imágenes, se aumentaría la calidad de estas imágenes. La principal limitación de esta mejora sería la cantidad de memoria disponible en la FPGA. Aumentar las dimensiones de las imágenes, implica una mayor cantidad de neuronas por lo que será necesario mayor espacio de almacenamiento, posiblemente siendo necesario escoger una FPGA diferente a la utilizada para la elaboración de este proyecto.

- **Reconocer imágenes mediante una cámara:** Si se consiguiese vincular una pequeña videocámara como una webcam a la FPGA, sería posible reconocer los números representados en la imagen proporcionada por la cámara. Esto nos permitiría realizar un reconocimiento de los distintos dígitos en tiempo real. Así mismo, permitiría a los distintos usuarios dibujar el número en un papel y la red sería capaz de distinguirlos.
  Esta mejora se encuentra limitada por la resolución de la imagen, por muy pequeña que sea la imagen proporcionada por la cámara, siempre tendrá unas dimensiones mayores que 28x28 píxeles.

Anexo 1: Script Matlab. Creación archivo .coe

```
for i=1:784                                          %save the selected image from the MNIST database
input(i)=testX(1,i);
end
%loop to convert all the input data
for i=1:784
x=single(input(i));                                  %convert double into single
y=bin2hex(ieee754(x));                               %convert into ieee754 standard and hexadecimal
result(i)= convertCharsToStrings(y);                 %convert character into string
end

fid=fopen('input.coe','wt');                         %open and create .coe file
fprintf(fid,'memory_initialization_radix=16;\n');    %write the data power
fprintf(fid,'memory_initialization_vector= \n');
for i=1:784
fprintf(fid,result(i));                              %write the result
fprintf(fid,',\n');                                  %write a ,
end
fclose(fid);                                         %close text file
```

Anexo 2: Implantación software red neuronal

```
function [y1] = myNeuralNetworkFunction(x1)
%MYNEURALNETWORKFUNCTION neural network simulation function.
%
% Generated by Neural Network Toolbox function genFunction, 01-Mar-2019 11:30:35.
%
% [y1] = myNeuralNetworkFunction(x1) takes these arguments:
%   x = Qx784 matrix, input #1
% and returns:
%   y = Qx10 matrix, output #1
% where Q is the number of samples.

%#ok<*RPMT0>

% ===== NEURAL NETWORK CONSTANTS =====

% Input 1
x1_step1.keep = [13 14 15 16 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 59 60 61 62 63 64 65 66 67 68 69 70
x1_step2.xoffset = [0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;
x1_step2.gain = [0.0172413793103448;0.0078740157480315;0.00925925925925926;0.222222222222222;0.125;0.0425531914893617;0.01273
x1_step2.ymin = -1;

% Layer 1
b1 = [1.3467925742778311;-1.0893886571190488;0.7658400540582585;0.45337510473090115;-0.15417577800431947;-0.18163866426062983;
IW1_1 = [0.049239900252351733 0.13081898008235943 0.060871287409910838 0.010417667332444144 0.055874123610462963 0.13368795630

% Layer 2
b2 = [-1.07345262178913;-0.71058123401006412;0.10397252219661689;0.062635209545928272;-1.0127192615348637;-0.034068958841080
LW2_1 = [0.13536514839587122 0.86458514429750544 1.4967564827084587 1.6816062356227846 1.6568346961417977 -0.70022084889630665


% ===== SIMULATION ========

% Dimensions
Q = size(x1,1); % samples

% Input 1
x1 = x1';
xp1 = removeconstantrows_apply(x1,x1_step1);
xp1 = mapminmax_apply(xp1,x1_step2);

% Layer 1
a1 = tansig_apply(repmat(b1,1,Q) + IW1_1*xp1);

% Layer 2
a2 = softmax_apply(repmat(b2,1,Q) + LW2_1*a1);

% Output 1
y1 = a2;
y1 = y1';
end


  % ===== MODULE FUNCTIONS ========

  % Map Minimum and Maximum Input Processing Function
  function y = mapminmax_apply(x,settings)
  y = bsxfun(@minus,x,settings.xoffset);
  y = bsxfun(@times,y,settings.gain);
  y = bsxfun(@plus,y,settings.ymin);
  end

  % Remove Constants Input Processing Function
  function y = removeconstantrows_apply(x,settings)
  y = x(settings.keep,:);
  end
```

```
% Competitive Soft Transfer Function
function a = softmax_apply(n,~)
if isa(n,'gpuArray')
    a = iSoftmaxApplyGPU(n);
else
    a = iSoftmaxApplyCPU(n);
end
end
function a = iSoftmaxApplyCPU(n)
nmax = max(n,[],1);
n = bsxfun(@minus,n,nmax);
numerator = exp(n);
denominator = sum(numerator,1);
denominator(denominator == 0) = 1;
a = bsxfun(@rdivide,numerator,denominator);
end
function a = iSoftmaxApplyGPU(n)
nmax = max(n,[],1);
numerator = arrayfun(@iSoftmaxApplyGPUHelper1,n,nmax);
denominator = sum(numerator,1);
a = arrayfun(@iSoftmaxApplyGPUHelper2,numerator,denominator);
end

function numerator = iSoftmaxApplyGPUHelper1(n,nmax)
numerator = exp(n - nmax);
end
function a = iSoftmaxApplyGPUHelper2(numerator,denominator)
if (denominator == 0)
    a = numerator;
else
    a = numerator ./ denominator;
end
end

% Sigmoid Symmetric Transfer Function
function a = tansig_apply(n,~)
a = 2 ./ (1 + exp(-2*n)) - 1;
end
```

Anexo 3: Código VHDL. Controlador memoria capa de entrada

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity controller_input is
    Generic ( size:integer ;                                    --size of adress (bits)
              value:integer );                                  --maximum count value
    Port  (clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           enable_input : in STD_LOGIC;                         --enable from floating_point_0 (result_tvalid)
           enable_weight:out std_logic;                         --connected to controller_weight (enable_weight)
           enable_hidden:out std_logic;                         --connected to controller_hidden (enable_hidden)
           enable_mux_input: out std_logic;                     --connected to mux_input (enable_hidden)
           adress_input:out std_logic_vector (size downto 0));  --adress memory_input

end controller_input;


architecture Behavioral of controller_input is
signal count: unsigned  (size downto 0);
begin
Process(clk,reset)
    begin
  if reset='1' then
    count<=(others=>'0');
    enable_weight<='0';
    enable_hidden<='0';
    enable_mux_input<='0';
  elsif clk'event and clk='1' then
    if enable_input='1' then
        if count= value then
            count<=(others=>'0');
            enable_weight<='0';
            enable_hidden<='1';
            enable_mux_input<='1';
        else
            count<=count +1;
            enable_weight<='1';
            enable_hidden<='0';
        end if;
    else
        enable_weight<='0';
        enable_hidden<='0';
    end if;
  end if;
end process;
adress_input<=std_logic_vector(count);

end Behavioral;
```

Anexo 4: Código VHDL. Controlador memoria pesos

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity controller_weight is
    Generic ( size:integer ;                                        --size of adress (bits)
              value:integer );                                      --maximum count value
    Port  (clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           enable_weight : in STD_LOGIC;                            --enable from controller input
           enable_weight2: in std_logic;                            --enable from controller hidden
           enable_weight3: in std_logic;                            --enable from controller hidden
           start:in std_logic;                                      --begging signal from edge detector
           enable_weight4: in std_logic;                            --enable from controller hidden
           enable_weight5: in std_logic;                            --enable from memory_mux_out (enable_output)
           valid:out std_logic;                                     --connected to valid_in (mux_input)
           valid2: out std_logic;                                   --coneected to tvalid floating_point_0
           adress_weight:out std_logic_vector (size downto 0));     --weight memory adress

end controller_weight;


architecture Behavioral of controller_weight is
signal count: unsigned  (size downto 0);
begin
Process(clk,reset)
    begin
  if reset='1' then
    count<=(others=>'0');
    valid<='0';
    valid2<='0';
  elsif clk'event and clk='1' then
    if enable_weight='1' or enable_weight4='1' then
        if count= value then
            count<=(others=>'0');
            valid<='0';
        else
            count<=count +1;
            valid<='1';
        end if;
    elsif start='1' then
        valid<='1';
    else
        valid<='0';
    end if;
```

```
    if enable_weight2='1' or enable_weight3='1' or enable_weight5='1' then
        if count=value then
            count<=(others=>'0');
            valid2<='0';
        else
            count<=count+1;
            valid2<='1';
        end if;
    else
        valid2<='0';
    end if;
  end if;
end process;
adress_weight<=std_logic_vector(count);

end Behavioral;
```

Anexo 5: Código VHDL. Controlador memoria capa oculta

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity controller_hidden is
    Generic ( size:integer ;                                        --size of adress (bits)
              value:integer );                                      --maximum count value
    Port  (clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           enable_hidden : in STD_LOGIC;                            --enable from multiplexor_memory_hidden (enable_hidden)
           enable_hidden2: in std_logic;                            --enable from floating_point_1 (result_tvalid)
           enable_output: out std_logic;                            --connected to multiplexor_1 (enable_output)
           enable_weight2:out std_logic;                            --connected to controller_weight (enable_weight2)
           enable_weight3:out std_logic;                            --connected to controller_weight (enable_weight3)
           enable_weight4:out std_logic;                            --connected to controller_weight (enable_weight4)
           adress_hidden:out std_logic_vector (size downto 0));     --adress memory hidden

end controller_hidden;


architecture Behavioral of controller_hidden is
signal count: unsigned  (size downto 0);
begin
Process(clk,reset)
    begin
  if reset='1'then
    count<=(others=>'0');
    enable_output<='0';
    enable_weight2<='0';
    enable_weight3<='0';
    enable_weight4<='0';
  elsif clk'event and clk='1' then
    if enable_hidden='1' then
        if count=value then
            count<=(others=>'0');
            enable_weight2<='1';
            enable_weight4<='0';
        else
            count<=count+1;
            enable_weight2<='0';
            enable_weight4<='1';
        end if;
    else
        enable_weight2<='0';
        enable_weight4<='0';
    end if;

    if enable_hidden2='1' then
        if count=value then
            count<=(others=>'0');
            enable_output<='1';
            enable_weight3<='0';
        else
            count<=count+1;
            enable_output<='0';
            enable_weight3<='1';
        end if;
    else
        enable_output<='0';
        enable_weight3<='0';
    end if;
 end if;
end process;
adress_hidden<=std_logic_vector(count);

end Behavioral;
```

Anexo 6: Código VHDL. Controlador memoria capa salida

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity controller_outputs is
    Generic ( size:integer ;                                        --size of adress (bits)
              value:integer );                                      --maximum count value
    Port  (clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           enable_output : in STD_LOGIC;                            --enable from multiplexor_memory_out (enable_output)
           enable_final: in std_logic;                              --enable from multiplexor_memory_out (enable_final)
           start: out std_logic;                                    --connected to mux_final (start)
           enable_softmax: out std_logic;                           --connected to mux_final (softmax)
           adress_output:out std_logic_vector (size downto 0));     --adress memory output

end controller_outputs;


architecture Behavioral of controller_outputs is
signal count: unsigned  (size downto 0);
begin
Process(clk,reset)
    begin
  if reset='1' then
    count<=(others=>'0');
    start<='0';
    enable_softmax<='0';
  elsif clk'event and clk='1' then
    if enable_output='1' then
        if count= value then
            count<=(others=>'0');
            start<='1';
        else
            count<=count +1;
            start<='0';
        end if;
    else
        start<='0';
    end if;

    if enable_final='1' then
        if count=value then
            count<=(others=>'0');
            enable_softmax<='0';
        else
            count<=count+1;
            enable_softmax<='1';
        end if;
     else
        enable_softmax<='0';
     end if;
  end if;
end process;
adress_output<=std_logic_vector(count);

end Behavioral;
```

Anexo 7: Código VHDL. Multiplexor capa oculta

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity multiplexor_memory_hidden is
    Port ( din : in STD_LOGIC_VECTOR (31 downto 0);                --data from floating_point_0
           din_activation : in STD_LOGIC_VECTOR (31 downto 0);     --data from activation function (result floating_point_9)
           din_bias: in std_logic_VECTOR (31 downto 0);            --data from floating_point_2
           valid_activation : in STD_LOGIC;                        --signal from floating_point_9 (result_tvalid)
           valid_bias: in std_logic;                               --signal from floating_point_2 (result_tvalid)
           we_normal : in STD_LOGIC;                               --signal from faloting_point_0 (result_tvalid)
           we_activation : in STD_LOGIC;                           --signal from floating_point_9 (result_tvalid)
           we_bias: in std_logic;                                  --signal from faloting_point_2 (result_tvalid)
           clk: in std_logic;
           reset: in std_logic;
           we : out STD_LOGIC;                                     --connected to memory output   (we)
           dout : out STD_LOGIC_VECTOR (31 downto 0);              --connected to memory output  (data)
           enable_activation_hidden: out std_logic;                --connected to floating_point_8 (tvalid)
           enable_hidden:out std_logic);                           --connected to counter hidden (enable_hidden)
end multiplexor_memory_hidden;


architecture Behavioral of multiplexor_memory_hidden is

begin
Process(valid_activation,valid_bias,we_normal,we_activation,we_bias,din,din_activation,din_bias)
begin
    if valid_activation='1' then
        dout<=din_activation;
        we<=we_activation;
    elsif valid_bias='1' then
        dout<=din_bias;
        we<=we_bias;
    else
        dout<=din;
        we<=we_normal;
    end if;
end process;
Process(reset,clk)
begin
    if reset='1' then
        enable_hidden<='0';
    elsif clk'event and clk='1' then
        if valid_activation='1' then
            enable_hidden<='1';
        else
            enable_hidden<='0';
        end if;
    end if;
end process;

Process(reset,clk)
begin
    if reset='1' then
        enable_activation_hidden<='0';
    elsif clk'event and clk='1' then
        if valid_bias='1' then
            enable_activation_hidden<='1';
        else
            enable_activation_hidden<='0';
        end if;
    end if;
end process;
end Behavioral;
```

Anexo 8: Código VHDL. Multiplexor memoria capa de salida

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity multiplexor_memory_out is
    Port ( din : in STD_LOGIC_VECTOR (31 downto 0);                 --data from floating_point_1
           din_activation : in STD_LOGIC_VECTOR (31 downto 0);      --data from activation function (result floating_point_7)
           din_bias: in STD_LOGIC_VECTOR (31 downto 0);             --data from floating_point_2
           din_softmax: in std_logic_vector(31 downto 0);           --data from floating_point_11
           valid_activation:in std_logic;                           --signal from floating_point_7 (result_tvalid)
           valid_bias: in std_logic;                                --signal from floating_point_2 (result_tvalid)
           valid_softmax:in std_logic;                              --signal from floating_point_11 (result_tvalid)
           we_normal : in STD_LOGIC;                                --signal from floating_point_1 (result_tvalid)
           we_activation : in STD_LOGIC;                            --signal from floating_point_7 (result_tvalid)
           we_bias:in std_logic;                                    --signal from floating_point_2 (result_tvalid)
           we_softmax: in std_logic;                                --signal from floating_point_11 (result_tvalid)
           clk: in std_logic;
           reset: in std_logic;
           enable_output: out std_logic;                            --connected to enable_output(controller_output)
           enable_activation_output: out std_logic;                 --connected to tvalid (floating_point_7)
           enable_final:out std_logic;                              --connected to enable_final (controller_output)
           we_out : out STD_LOGIC;                                  --connected to we (memory_output)
           dout : out STD_LOGIC_VECTOR (31 downto 0));              --connected to d (memory_output)
end multiplexor_memory_out;


architecture Behavioral of multiplexor_memory_out is

begin
Process(valid_activation,valid_bias,we_normal,we_activation,we_bias,din,din_activation,din_bias,din_softmax,we_softmax,valid_softmax)
begin
    if valid_activation='1' then
        we_out<=we_activation;
        dout<=din_activation;
    elsif valid_bias='1' then
        we_out<=we_bias;
        dout<=din_bias;
    elsif valid_softmax='1' then
        we_out<=we_softmax;
        dout<=din_softmax;
    else
        we_out<=we_normal;
        dout<=din;
    end if;
end process;
Process(clk,reset)
begin
    if reset='1' then
        enable_output<='0';
    elsif clk'event and clk='1' then
        if valid_activation='1' then
            enable_output<='1';
        else
            enable_output<='0';
        end if;
    end if;
end process;
```

```
Process(clk,reset)
begin
    if reset='1' then
        enable_activation_output<='0';
    elsif clk'event and clk='1' then
        if valid_bias='1' then
            enable_activation_output<='1';
        else
            enable_activation_output<='0';
        end if;
    end if;
end process;
Process(clk,reset)
begin
    if reset='1' then
        enable_final<='0';
    elsif clk'event and clk='1' then
        if valid_softmax='1' then
            enable_final<='1';
        else
            enable_final<='0';
        end if;
     end if;
end process;

end Behavioral;
```

Anexo 9: Código VHDL. Multiplexor función de activación

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity multiplexor is
    Port ( clk: in std_logic;
           reset: in std_logic;
           enable_hidden : in STD_LOGIC;                    --signal from controller_input(enable_hidden)
           enable_output : in STD_LOGIC;                    --signal from controller_input(enable_output)
           din_hidden : in STD_LOGIC_VECTOR (31 downto 0);  --data from memory hidden
           din_output : in STD_LOGIC_VECTOR (31 downto 0);  --data from memory output
           valid_bias: out std_logic;                       --connected to valid (floating_point_12)
           identifier:out std_logic;                        --connected to valid (multiplexor_output)
           dout : out STD_LOGIC_VECTOR (31 downto 0));      --conected to floating_point_12
end multiplexor;


architecture Behavioral of multiplexor is

begin
Process(reset,clk)
    begin
    if reset='1' then
        dout<=(others=>'0');
        valid_bias<='0';
        identifier<='0';
    elsif clk'event and clk='1' then
        if enable_hidden='1' then
            dout<=din_hidden;
            valid_bias<='1';
            identifier<='0';                            --0 when data is form hidden
        elsif enable_output='1' then
            dout<=din_output;
            valid_bias<='1';
            identifier<='1';                            --1 when the data is from output
        else
            valid_bias<='0';
            dout<=(others=>'0');
        end if;
     end if;
end process;
end Behavioral;
```

Anexo 10: Código VHDL. Módulo Led

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;


entity led is
    Port ( clk: in std_logic;
           reset: in std_logic;
           din : in STD_LOGIC_vector(7 downto 0);   --data from memory_output
           valid: in std_logic;                     --signal from floating_point6 (tvalid)
           led0 : out STD_LOGIC;
           led1 : out STD_LOGIC;
           led2 : out STD_LOGIC;
           led3 : out STD_LOGIC);
end led;


architecture Behavioral of led is
signal count: integer range 0 to 10;               --internal signal counter
signal data:std_logic_vector(9 downto 0);          --1 in the position of the neuron greater than 0.6, 0 if not
signal ok: std_logic;                              --1 when all the bits of data are set
begin
Process(reset,clk)
begin
    if reset='1' then
        count<=0;
        ok<='0';
        data<=(others=>'0');
    elsif clk'event and clk='1' then
        if valid='1' then
            data(count)<=din(0);
            if count=9 then
                count<=0;
                ok<='1';
            else
                count<=count +1;
                ok<='0';
            end if;
        end if;
    end if;
end process;
```

```
Process(ok,data)
begin
led3<='0'; led2<='0'; led1<='0'; led0<='0';
 if ok='1' then
    case data is
        when "0000000000" =>
        when "0000000001" => led3<='1'; led2<='1'; led1<='1'; led0<='1';
        when "0000000010" => led0<='1';
        when "0000000100" => led1<='1';
        when "0000001000" => led1<='1'; led0<='1';
        when "0000010000" => led2<='1';
        when "0000100000" => led2<='1'; led0<='1';
        when "0001000000" => led2<='1'; led1<='1';
        when "0010000000" => led2<='1'; led1<='1'; led0<='1';
        when "0100000000" => led3<='1';
        when "1000000000" => led3<='1'; led0<='1';
        when others=> led3<='0'; led2<='0'; led1<='0'; led0<='0';
    end case;
  else
    led3<='0'; led2<='0'; led1<='0'; led0<='0';
  end if;
end process;

end Behavioral;
```

Anexo 11: Código VHDL. Módulo Look Up Table

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


use IEEE.NUMERIC_STD.ALL;


entity lookup is
    Port ( din : in STD_LOGIC_VECTOR (15 downto 0);          --data from floating_point_8 (fixed to float transform)
           valid : in STD_LOGIC;                             --signal from floating_point_8 (tvalid)
           ok : out STD_LOGIC;                               -- connnected to valid_activation and ve_activation (multiplexor_memory_hidden)
           dout : out STD_LOGIC_VECTOR (31 downto 0));       --connected to d (memory_hidden)
end lookup;


architecture Behavioral of lookup is

begin
Process(valid,din)
begin
    if valid='1' then
        ok<='1';
       if  unsigned(din)> 6 and unsigned(din)< x"8000" then      -- check if din>6 and positive
            dout<=x"3f800000";
       else
        case din is
            when x"fffb"=> dout<=x"bf800000";                    -- fffb= -5
            when x"fffc"=> dout<=x"bf7fd40c";                    -- fffc= -4
            when x"fffd"=> dout<=x"bf7ebbe9";                    -- fffd= -3
            when x"fffe"=> dout<=x"bf76ca83";                    -- fffc= -2
            when x"ffff"=> dout<=x"bf42f7d6";                    -- ffff=-1
            when x"0000"=> dout<=(others=>'0');
            when x"0001"=> dout<=x"3f42f7d6";
            when x"0002"=> dout<=x"3f76ca83";
            when x"0003"=> dout<=x"3f7ebbe9";
            when x"0004"=> dout<=x"3f7fd40c";
            when x"0005"=> dout<=x"3f7ffa0d";
            when x"0006"=> dout<=x"3f800000";
            when others=> dout<=x"bf800000";
         end case;
      end if;
      else
        ok<='0';
        dout<=(others=>'0');
      end if;
end process;
end Behavioral;
```

Bibliografía

[1] B. Yegnanarayana, Artificial Neural Networks, New Delhi: Prentice-Hall of India Pvt.Ltd, 1999.

[2] B. D. S.L, «GPU vs FPGA Performance Comparison,» Berten DSP S.L, 2016.

[3] R. Rojas, Neural Networks. A Systematic Introduction, Berlin: Springer - Verlag, 1996.

[4] E. B. Zuleta, El sistema nervioso. Desde las neuronas hasta el cerebro humano, Colombia: Universidad de Antioquia, 2004.

[5] M. Nielsen, «Neural Networks and Deep Learning,» Determination Press, 2015. [En línea]. Available: http://neuralnetworksanddeeplearning.com/chap1.html.

[6] S. Sharma, «Activation Functions in Neural Networks,» Towards Data Science, 2017. [En línea]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[7] A. Tch, «The mostly complete chart of Neural Netwoks,» 2017. [En línea]. Available: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464.

[8] D. Calvo, «Red Neuronal Convolucional CNN,» 2017. [En línea]. Available: http://www.diegocalvo.es/red-neuronal-convolucional/.

[9] D. Britz, «Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs,» WILDML - Artificial Intelligence, Deep Learning and NLP , 2015. [En línea]. Available: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

[10] H. Sak, F. Beaufays y A. Senior, «Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling,» de *InterSpeech*, Singapore, 2014.

[11] J. Chung, C. Gulcehre, K. Cho y Y. Bengio, «Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,» NIPS, Montréal, 2014.

[12] A. Ballesteros y D. E. Dominguez, «Neural Networks Framework,» Universidad de Málaga, [En línea]. Available: http://www.redes-neuronales.com.es/tutorial-redes-neuronales/clasificacion-de-redes-neuronales-respecto-al-aprendizaje.htm.

[13] A. Delgado, «Aplicación de las Redes Neuronales en Medicina,» *Revista de la Facultad de Medicina - Universidad Nacional de Colombia,* vol. 47, nº 4, pp. 221-223, 1999.

[14] X. B. Olabe, «Redes Neuronales Artificiales y sus Aplicaciones,» Escuela Superior de Ingeniería de Bilbao, EHU, Bilbao, 1998.

[15] A. R. Omondi y J. C. Rajapakse, FPGA Implementation of Neural Networks, Países Bajos: Springer, 2006.

[16] S. Karsoliya, «Approximating Number of Hidden layer neurons in Multiple Hidden Layer BPNN Architecture,» *International Journal of Engineering Trends and Technology*, vol. 3, nº 6, pp. 714-717, 2012.

[17] S. W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing, California: California Technical Publishing, 1997.